

从技术原理到编码实践
完整介绍以太坊DApp的开发流程与实践
涵盖钱包、交易所与智能合约的开发与部署

# 区块链 以太坊DApp开发实战

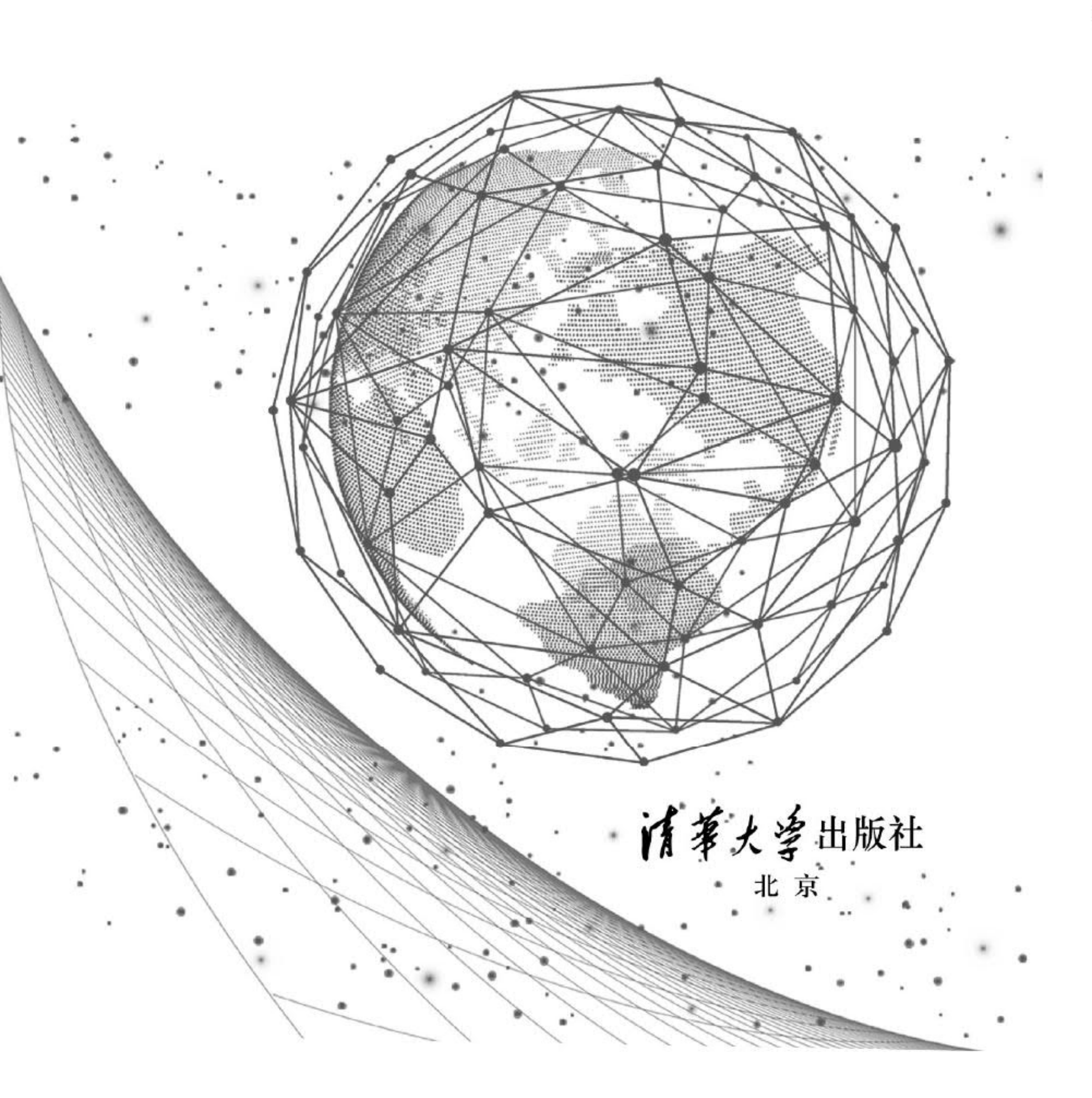
林冠宏 ◎ 著

清華大學出版社

# 区块链

# 以太坊DApp开发实战

林冠宏 ◎ 著



清華大學出版社
北京

## 内 容 简 介

本书以 Go 编程语言讲解，从必要的理论知识到编码实践，循序渐进地介绍以太坊 DApp 开发的技术要点。全书主要内容分 4 大部分：第一部分介绍区块链的一些重要基础知识；第二部分全面地介绍以太坊公链的应用基础，内容包含但不限于以太坊的大量术语；第三部分介绍以太坊智能合约整体开发与部署实践；第四部分以以太坊 DApp 中继服务作为综合范例，介绍以太坊区块链 DApp 的开发流程与实战。

本书注重应用，代码注释详尽，适合 IT 技术开发者阅读，对于想了解以太坊及其 DApp 开发技术的人员或想开发一款基于以太坊的 DApp 开发者尤为合适。



**图书在版编目（CIP）数据**

区块链以太坊 DApp 开发实战 / 林冠宏著. —北京：清华大学出版社，2019
ISBN 978-7-302-53126-5

Ⅰ.①区…　Ⅱ.①林…　Ⅲ.①电子商务－支付方式－程序设计　Ⅳ.①F713.361.3②TP311.1

中国版本图书馆 CIP 数据核字（2019）第 112868 号

**责任编辑：**王金柱
**封面设计：**王　翔
**责任校对：**闫秀华
**责任印制：**杨　艳

**出版发行：**清华大学出版社
**网　　址：**http://www.tup.com.cn，http://www.wqbook.com
**地　　址：**北京清华大学学研大厦 A 座　　**邮　　编：**100084
**社 总 机：**010-62770175　　**邮　　购：**010-62786544
**投稿与读者服务：**010-62776969，c-service@tup.tsinghua.edu.cn
**质 量 反 馈：**010-62772015，zhiliang@tup.tsinghua.edu.cn

**印 装 者：**北京密云胶印厂
**经　　销：**全国新华书店
**开　　本：**190mm×260mm　　**印　　张：**16　　**字　　数：**409 千字
**版　　次：**2019 年 8 月第 1 版　　**印　　次：**2019 年 8 月第 1 次印刷
**定　　价：**79.00 元

---

产品编号：082466-01

# 前　言

出版这本书之前，我是一位已经撰写技术博文 5 年之久的程序员，技术栈范围包含但不限于安卓应用开发、后端开发以及现在所从业的区块链 DApp 开发。

曾经有不少出版社联系我出书，但限于对知识的敬畏和对出书的谨慎，都一一婉拒了。正式签约出版这本书是 2018 年 10 月中旬，那时我处于一个对区块链和以太坊知识非常热衷的阶段，当时的工作也正好是基于区块链做各种 DApp 的开发，比如具有代表性的钱包、中心化交易所和去中心化交易所应用。对区块链、以太坊的各个方面构建了一套完整的知识体系，所以在清华大学出版社的编辑联系我的时候，市面上关于以太坊 DApp 技术开发的书籍几乎为零，而理论性的书籍过多，在深思熟虑之后，便决定编写此书。

写书最怕的是误人子弟。后面正式编写的时候才发现，将整个以太坊的知识体系展开来讲的话，有很多的细节是自己之前还没有掌握的，比如：区块链浏览器上所看到的非 ETH 交易记录不能作为资产转移成功的依据等。编写此书的过程中，也遇到了一些疑惑点，通过借鉴优秀的博客文章、阅读源码和咨询业界一些技术大佬的意见，反复检查、检验整理编写入书内，对我自己来说也是一种提升，丰富并拓展了我的以太坊知识体系。我在这里衷心感谢他们并将会在书后列举出这些文章的链接和相关大佬的名字。

全书的内容关联性很强，篇幅适中，非必要的理论性内容几乎没有谈及。在术语阐述上，我尽力做到用通俗的语言去讲解，如果读者在阅读的过程中依然无法理解某一个知识点，欢迎通过我的联系方式直接询问，我会为你们一一解答。

虽然，笔者已尽最大努力避免书中内容出现错误，但由于水平所限，难免会有错误，如果读者在书中发现错误的结论，欢迎联系我进行勘误。我将十分感谢您！对于被纠正的内容，我将会在技术博客中公布。此外，技术交流方面可以加入 QQ 群，群号等联系信息参见本书的后记。

林冠宏

2019 年 3 月

# 目　　录

# 第 1 章

# 区块链基础知识准备

本章我们将首先从区块链的基本概念入手，逐步介绍共识机制、共识算法、链的分叉等概念，以帮助读者建立有关区块链的知识体系，为后续的开发工作做好准备。

## 1.1　认识区块链

### 1.1.1　区块链的概念

我们一般意识形态中的链是铁链，由铁铸成，一环扣一环。区块链也可以这么理解，只不过它不是由铁铸成，而是由拥有一定数据结构的块连接而成，呈链状结构，这种结构就是链表。

区块抽象到计算机语言中就是一个对象、一个结构体、一个类，同样类中也可以定义属性、变量和方法，但区块里包括的内容可以自己来定义。比如，以太坊公链的区块结构，它有变量，我们就可以自己进行定义。以下是我们设置一个区块包括变量的例子。

```
type Block struct {
    Number   string  // 区块号
    PreHash  string  // 前一个区块的哈希值
    Hash     string  // 自身的哈希值
    Value    string  // 携带的数据
    Create   int64   // 创建的时间戳
}
```

上述的 type Block struct 表示定义一个区块，其中定义了变量 Number、PreHash、Hash、Value、Create。

当链表中的每个数据个体是上述区块的时候就构成了一条区块链。区块是区块链每一环的实体。这是一种最简单的区块链。如图 1-1 所示，其中箭头的方向代表的是子块关联父块，也可以将

箭头反过来，表示父块连接子块。

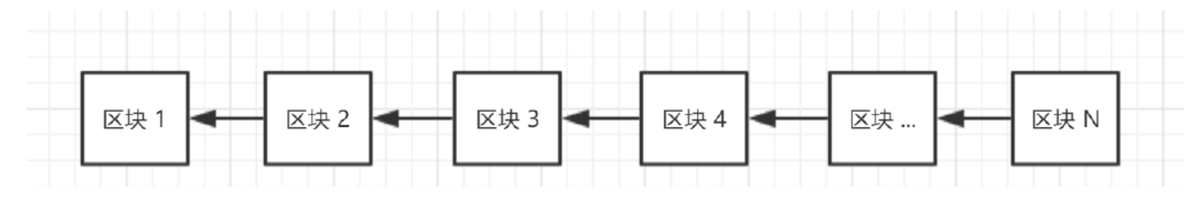



图 1-1　正常形态的链

由于链中的区块包含数据，例如上面的 Value 变量，因此我们能够在这个区块被打包到链中的时候向 Value 填充值，此后我们通过访问这个区块内部的数据可对它打包的数据进行读取，然后输出，展示给用户。

在上面的例子中，我们用来存储打包到区块中的数据变量只有一个 Value，那么请想象一下，如果把 Value 换成一个数组或者更多变量，这个区块就会变得更复杂，它的功能也会跟着变得更多。

此外，链中的区块被规定是唯一的，即相同区块号的区块不能以同一个身份（以太坊中允许有区块号一样的不同含义块）在同一条链中出现两次，如果出现了，那么链会将其纠正过来。

下面是网上对区块链的定义解释：

“区块链是分布式数据存储、点对点传输、共识机制、加密算法等计算机技术的新型应用模式。”

这个概念其实是一个广义的解释，笔者更趋向于把这个解释理解为区块链节点程序，而不是区块链，因为一个区块链的节点服务程序就包含了这个概念中的各个模块，实际上还有很多其他的模块。

一般来说，区块链公链包含但不限于下面的技术模块：

（1）数据加密签名技术模块。
（2）共识机制技术模块。
（3）分布式数据存储技术模块。
（4）点对点通信传输技术模块。
（5）智能合约技术模块。
（6）应用程序接口技术模块。

当我们把这些模块技术实现的代码整合到一个程序中时，它便是一个区块链应用，例如某一条公链。

那么是不是区块链应用一定要全部实现这些技术模块呢？不是的，你可以开发自己的区块链公链，哪怕是超级简单的雏形，只要是链状的区块存储应用，就可以称为区块链。请记住，任何一个复杂的区块链应用，例如知名的公链，都是在简单的模型上进行技术的添砖加瓦打造出来的。此外，区块链的各个技术模块所包含的知识点也是非常丰富的，可以说每一个知识点都属于一个领域。

## 1.1.2　链的分类

区块链的链分类通常有 3 类，即公有链、私有链和联盟链。这 3 类链的主要区别是：

（1）公有链的维护节点比较多，节点网络对所有人开放，任何人都可以进行特定的数据访问。

（2）私有链是面向个人或某个组织的。

（3）联盟链是多个组织团体的节点联合在一起维护的，对组织开放。

目前被广泛接受、认可、有价值的“代币”（Token）几乎都是基于公有链的。

不同种类的公有链之间要实现相互通信，比如比特币公链和以太坊公链进行 BTC 兑换 ETH 的交易，需要借助技术手段来实现，例如跨链通信技术。

### 1.1.3　区块链能做什么

从区块链普遍的去中心化的特点来看，在节点网络中，如果某条公链的合法节点数目达到一定的数量级，那么我们可以认为当前公链的去中心化程度接近 100%，这意味着链上的数据不会再被篡改了，于是我们所传递到链上被保存在区块中的数据会一直存在下去，真实而永久。

基于这个特点，我们可以将区块链应用到数据的溯源存储方面。除此之外，还可以根据区块链具体提供的功能进行各种应用。例如，以太坊公链，它是区块链，而且提供了智能合约这类具备图灵完备的功能模块，我们可以基于它来开发智能合约去中心化应用 DApp，其中最为普遍的便是 ERC20 智能合约所对应的“代币”。

要理解区块链能做什么，可以从实际的区块链应用所具备的特点进行思考，从而得出答案。

## 1.2　共识的作用

每条区块链的节点，例如以太坊节点，都拥有自己存储数据的地方，节点之间虽然会相互通信，但又彼此不依赖，这是因为互不信任。

在这种情况下，各个节点如何保证在互相通信的过程中维护数据的一致性，从而使链上相同区块号的区块只有一个呢？此时就诞生了区块链技术栈中的另一个知识点：共识，又称共识机制。

所谓共识，通俗来讲，就是我们大家对某种事物的理解达成一致的意思。比如说日常开会讨论问题，又比如判断一个动物是不是猫，我们肉眼看了后觉得像猫，其符合猫的特征，那么我们认为它就是猫。这就是共识，可见共识是一种规则。

继续上述会议的例子。参与会议的人，通过开会的方式达到解决问题的目的。对比区块链中参与挖矿的节点，节点中有矿工这么一种角色，它在代码中对应某一个功能模块。节点矿工通过某种共识方式（算法）来解决该节点的账本与其他节点的账本保持一致。账本保持一致的意思是：各个节点同步的区块的信息保持一致，以维护同一条区块链。

那么为什么需要共识呢？没有共识可不可以？当然不可以，这样会出现问题，假如生活中没有共识规则，那么一切都会乱套。区块链与此类似，没了共识规则，各个节点各干各的，会失去一致性，区块链也不会达成统一。

上述会议和区块链的对应关系如下：

（1）参会的人=挖矿的矿工

（2）开会=共识方式（算法）

（3）讨论解决问题=让自己的账本跟其他节点的账本保持一致

你可能会对上面的内容产生一些疑问：

（1）区块链节点和矿工是什么关系？

（2）让节点账本保持一致，账本的内容是什么？

（3）为什么需要共识算法去保持账本一致？

首先，我们来看一下区块链节点和矿工的关系。矿工是区块链节点中的一个角色，从编程的角度来看，就是程序中的一个功能模块。因此可见，矿工与区块链就是包含与被包含的关系。

其次，让节点账本保持一致，账本的内容是什么？账本的内容就是所有节点所维护的那条公链中的区块以及该区块的相关信息。要保持这条链不出差错，块与块之间必须正常相连。

最后，为什么需要共识算法来保持账本一致呢？因为区块会被节点中的一些功能模块生成，在众多节点且相同的时间流逝中，A 节点有可能诞生一个区块 1，B 节点也有可能诞生一个区块 1，这样它们诞生的区块号就发生重复了。在同一条链中，相同区块号的区块最终只能挑选一个串接到链中，这时取谁的好呢？此时就需要用共识算法这一规则来做出选择了。这个选择的大致形式可参考图 1-2。

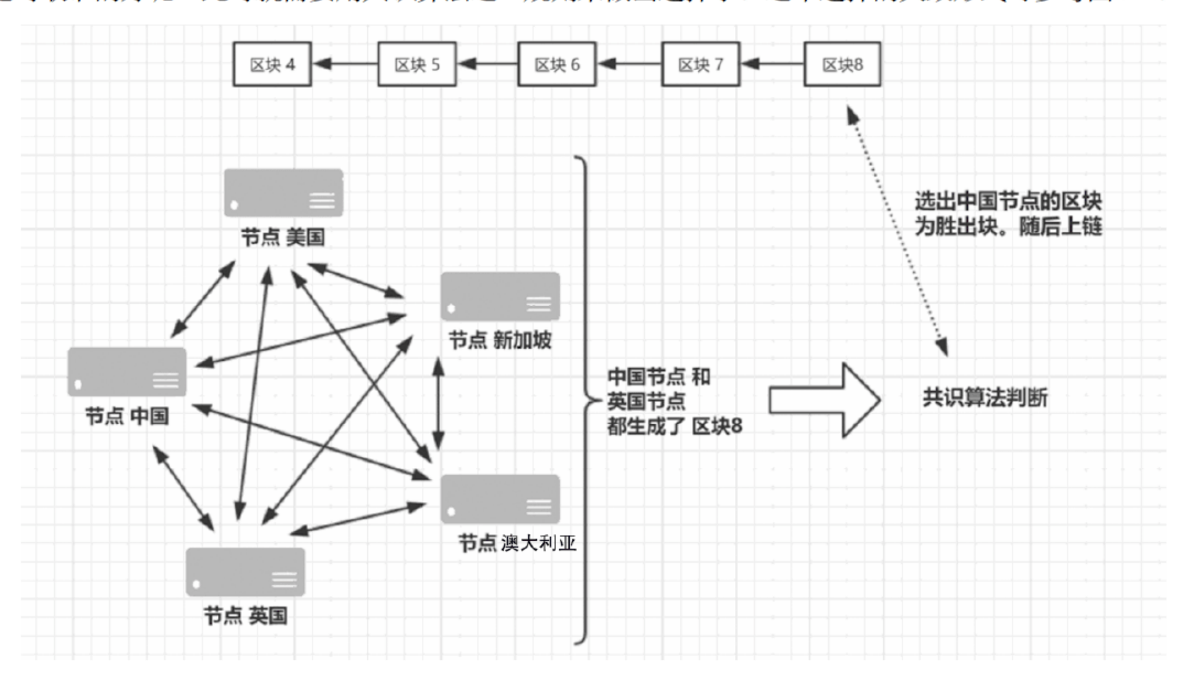



图 1-2　用共识算法选出胜出块

# 1.3　常见的共识算法

目前在区块链中，使节点账本保持一致的共识算法常见的有如下几种：

- PoW，代表者是比特币（BTC），区块链 1.0。
- PoS，代表者是以太坊（ETH），以太坊正在从 PoW 过渡到 PoS，区块链 2.0。

- DPoS，代表者是柚子（EOS），区块链 3.0。
- PBFT 拜占庭容错，联盟链中常用。

下面通俗地介绍前 3 种共识算法的概念及优缺点。

## 1.3.1　PoW 算法

PoW（Proof of Work，工作量证明）的字面意思是谁干的活多，谁的话语权就大，在一定层面上类似于现实生活中“多劳多得”的概念。

以比特币为例，比特币挖矿就是通过计算符合某一个比特币区块头的哈希散列值争夺记账权。这个过程需要通过大量的计算实现，简单理解就是挖矿者进行的计算量越大（工作量大），它尝试解答问题的次数也就变得越多，解出正确答案的概率自然越高，从而就有大概率获得记账权，即该矿工所挖出的区块被串接入主链。

下面对上述一段话所涉及的几个术语做一下解释。

（1）区块头（Header）：区块链中区块的头部。比如你有一个饭盒，饭盒的第一层类似动物头部，称之为头部；第一层放着米饭，米饭就是头部装载的东西。

（2）哈希（Hash）：数学中的散列函数（数学公式）。

（3）哈希散列值：通过哈希函数得出的值。例如，有加法公式“1+2=3”，那么哈希公式 hash(1,2) 计算出来的结果即为哈希散列值。

（4）区块头的哈希散列值：饭盒第一层装的是米饭，那么这个值就是区块头装的东西。

（5）记账权，话语权：在大家都参与挖区块的情况下，谁挖出的区块是有效的，谁就有记账权或话语权。

在 PoW 共识算法下，当很多个节点都在挖矿时，每个节点都有可能挖出一个区块。比特币区块链定义了区块被挖出后，随之要被广播到其他节点中去，然后每个节点根据对应的验证方式对区块进行是否合法的验证操作，被确认合法的区块便会被并入主链中去。

对比现实生活，比如数学竞赛，参赛者相当于矿工，一道题目，谁先做出就公布计算过程和答案，不由裁判判断，由参赛者来一起验证；大家都认可后，宣布该题目结束，解题者及相关信息被记录到纸质册子或数据库，之后继续下一道题。

回到比特币挖矿中，其实就是计算出正确的哈希散列值，一旦计算出来，就生成新区块，并将生成的区块信息以广播的形式告诉其他节点。其他节点收到广播信息后，停下手上的计算工作，开始验证该区块的信息。若信息有效，则当前最新区块被节点承认，各个节点开始挖下一个区块；若信息无效，则各个节点继续自己的计算工作。这里的难题在于哈希散列值的计算随着比特币中难度系数（一个能增加计算难度的变量数字）的增大会越来越困难，导致计算需要耗费大量的电力资源，工作量巨大。

此外，成功挖出有效区块的矿工（节点）将会获得奖励。在不同的区块链体系中，奖励的东西不同，比特币区块链体系中的奖励是比特币。

图 1-3 是节点使用 PoW 共识算法共同承认胜出块，并将胜出块串接上链的模型图。

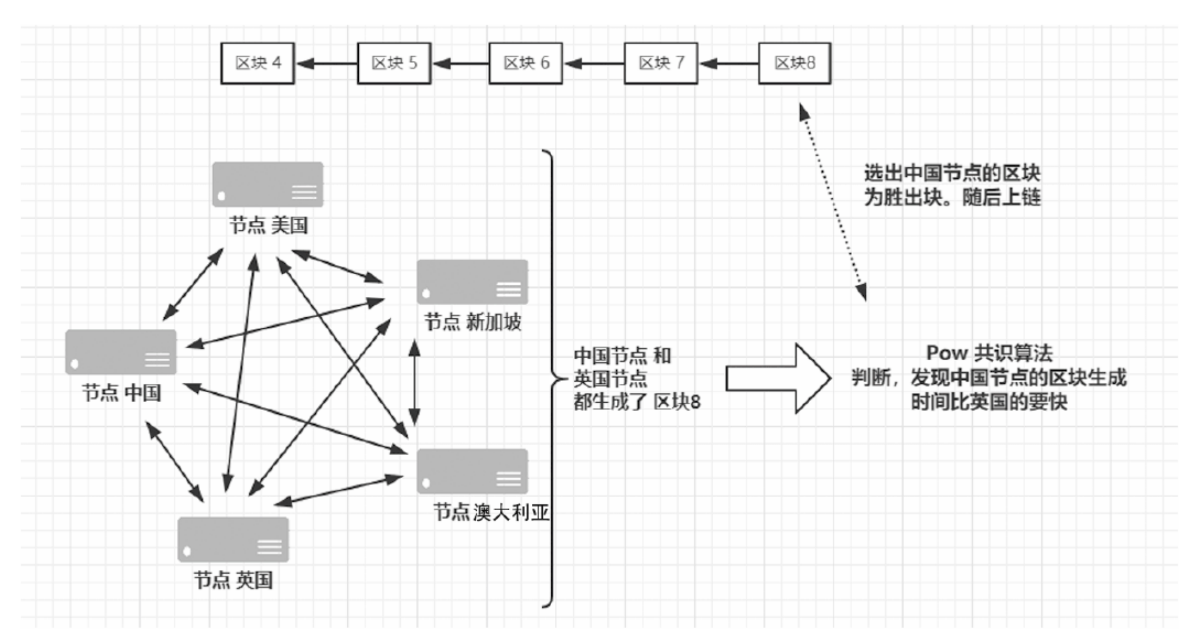



图 1-3 用 PoW 共识算法选出胜出块

PoW 共识算法具有下面的优缺点：

### （1）优点

- 机制设计独特。例如挖矿难度系数自动调整、区块奖励逐步减半等，这些因素都是基于经济学原理的，能吸引和鼓励更多的节点参与挖矿。
- 早参与早获利。越早参与的人获得的越多。在初始阶段，会促使加密货币迅速发展，节点网络迅速扩大。
- 通过“挖矿”的方式发行币，把代币分散给个人，实现了相对公平。

### （2）缺点

- 算力是计算机硬件（CPU、GPU 等）提供的，要耗费电力，是对能源的直接消耗，与人类追求节能、清洁、环保的理念相悖。
- PoW 机制发展到今天，算力的提供已经不再是单纯的 CPU 了，而是逐步发展到 GPU、FPGA 乃至 ASIC 矿机。用户也从个人挖矿发展到大的矿池、矿场，算力集中越来越明显。这与去中心化的思想背道而驰。
- 按照目前的挖矿速度，随着难度越来越大，当挖矿的成本高于挖矿收益时，人们挖矿的积极性会降低，造成大量算力减少。
- 基于 PoW 节点网络的安全性令人堪忧。
- 大于 51%算力的攻击。在“PoW 共识机制的 51%算力攻击”一节中会详细介绍。

问题解答：

（1）如果遇到同时解出问题的情况怎么办？

确实存在会同时解出的情况，即使我们把区块生成时间的时间戳定义到秒或者毫秒级别，依然会有同时间挖到矿的情况。对于这种情况，PoW 共识算法无能为力。具体的解决方法会在“链的分叉”一节中谈到。

（2）为什么是 51%算力，而不是 50.1%？

这个问题将会在“PoW 共识机制的 51%算力攻击”一节中进行解答。

## 1.3.2 PoS 算法

PoS（Proof of Stake，股权证明）是由点点币（PPCoin）首先应用的。该算法没有挖矿过程，而是在创世区块内写明股权分配比例，之后通过转让、交易的方式，也就是我们说的 IPO（Initial Public Offerings）公开募股方式，逐渐分散到用户钱包地址中去，并通过“利息”的方式新增货币，实现对节点地址的奖励。

PoS 的意思是股份制。也就是说，谁的股份多，谁的话语权就大，这和现实生活中股份制公司的股东差不多。但是，在区块链的应用中，我们不可能真实地给链中的节点分配股份，取而代之的是另外一些东西，例如代币，让这些东西来充当股份，再将这些东西分配给链中的各节点。下面我们通过示例来阐述这个概念。

例如，在虚拟货币的应用中，我们可以把持币量的多少看作拥有股权、股份的多少，假设某区块链公链使用了最基础的还没进行变种开发的 PoS 共识机制，以节点所拥有的 XXX 代币的数量来衡量这个节点拥有的股份是多少。假设共有 3 个节点，A、B 和 C，其中 A 节点拥有 10000 个 XXX 代币，而 B、C 节点分别有 1000 个、2000 个，那么在这个区块链网络中 A 节点产生的区块是最有可能被选中的，它的话语权是比较大的。

再例如，假设某条非虚拟货币相关的与实体业结合的公有链，汽车链，我们可以把每一位车主所拥有的车辆数目和他的车价值多少钱来分配股份（比如规定一个公式：车数×车价值=股份的多少）。

可见，在 PoS 中，股份只是一个衡量话语权的概念。我们可以在自己的 PoS 应用中进行更加复杂的实现，比如使用多个变量参与到股份值的计算中。

PoS 共识算法以拥有某样东西的数量来衡量话语权的多少，只要节点拥有这类东西，哪怕只拥有一个，也是有话语权的，即使这种话语权很小。

在 PoS 中，块是已经铸造好的。PoW 有挖矿的概念，而 PoS 没有。在 BTC 比特币公链中，可以挖矿；而在没有使用 PoS 共识算法的公链节点中，就没有挖矿这一回事。当然，如果将挖矿的概念进行其他拓展，则另当别论。

PoS 共识算法具有下面的优缺点：

### （1）优点

- 缩短了共识达成的时间，链中共识块的速度更快。
- 不再需要大量消耗能源挖矿，节能。
- 作弊得不偿失。如果一名持有多于 50% 以上股权的人（节点）作弊，相当于他坑了自己，因为他是拥有股权最多的人，作弊导致的结果往往是拥有股权越多的人损失越多。

### （2）缺点

- 攻击成本低，只要节点有物品数量，例如代币数量，就能发起脏数据的区块攻击。
- 初始的代币分配是通过 IPO 方式发行的，这就导致“少数人”（通常是开发者）获得了大量

成本极低的加密货币，在利益面前，很难保证这些人不会大量抛售。

- 拥有代币数量大的节点获得记账权的概率会更大，使得网络共识受少数富裕账户支配，从而失去公正性。

### 1.3.3 DPoS 算法

PoW 和 PoS 虽然都能在一定程度上有效地解决记账行为的一致性共识问题，也各有各的优缺点，但是现有的比特币 PoW 机制纯粹依赖算力，导致专业从事挖矿的矿工群体似乎已和比特币社区完全分隔，某些矿池的巨大算力俨然成为另一个中心，这与比特币的去中心化思想相冲突。PoS 机制虽然考虑到了 PoW 的不足，但依据 IPO 的方式发行代币数量，导致少部分账户代币量巨大，权力也很大，有可能支配记账权。DPoS（Delegated Proof of Stake，股份授权证明机制）共识算法的出现就是为了解决 PoW 和 PoS 的不足。

DPoS 引入了“见证者节点”这个概念。见证者节点可以生成区块。注意，这里有权限生成区块的是见证者节点，而不是持股节点。

下面我们主要以 EOS 区块链为例介绍 DPoS 算法。

持有 EOS 代币的节点为持股节点，但不一定是见证者节点。见证者节点由持股节点投票选举产生。DPoS 的选举方式如下：每一个持有股份的节点都可以投票选举见证者节点，得到总同意票数中的前 N 位候选者可以当选为见证者节点。这个 N 值需满足：至少一半的参与投票者相信 N 已经充分地去中心化（至少有一半参与投票的持股节点数认为，当达到了 N 位见证者的时候，这条区块链已经充分地去中心化了），且最好是奇数。请注意，最好是奇数的原因会在分叉一节中进行说明。

见证者节点的候选名单每个维护周期更新一次，见证者节点们被选出之后，会进行随机排列，每个见证者节点按顺序有一定的权限时间生成区块，若见证人在给定的时间片不能生成区块，区块生成权限将交给下一个时间片对应的见证人。DPoS 的这种设计使得区块的生成更为快速，也更加节能。这里“一定的权限时间”不受算法硬性限制。此外，见证者节点的排序是根据一定算法随机进行的。

DPoS 共识算法具有下面的优缺点：

**（1）优点**

- 能耗更低。DPoS 机制将节点数量进一步减少到 N 个，在保证网络安全的前提下，整个网络的能耗进一步降低，网络运行成本最低。
- 更加去中心化，选举的 N 值必须充分体现中心化。
- 避免了 PoS 的少部分账户代币量巨大导致权力太大的问题，话语权在被选举出的 N 个节点中。
- 更快的确认速度，由见证者节点进行确认，而不是所有的持股节点。

**（2）缺点**

- 投票的积极性并不高。绝大多数持股节点未参与投票。因为投票需要时间、精力等。
- 选举固定数量的见证人作为记账候选人有可能不适合完全去中心化的场景，在网络节点很少的场景，选举的见证人的代表性也不强。

- 对于坏节点的处理存在诸多困难。社区选举不能及时有效地阻止一些破坏节点的出现，给节点网络造成安全隐患。

图 1-4 所示是 DPoS 共识算法选举的大致模型图。

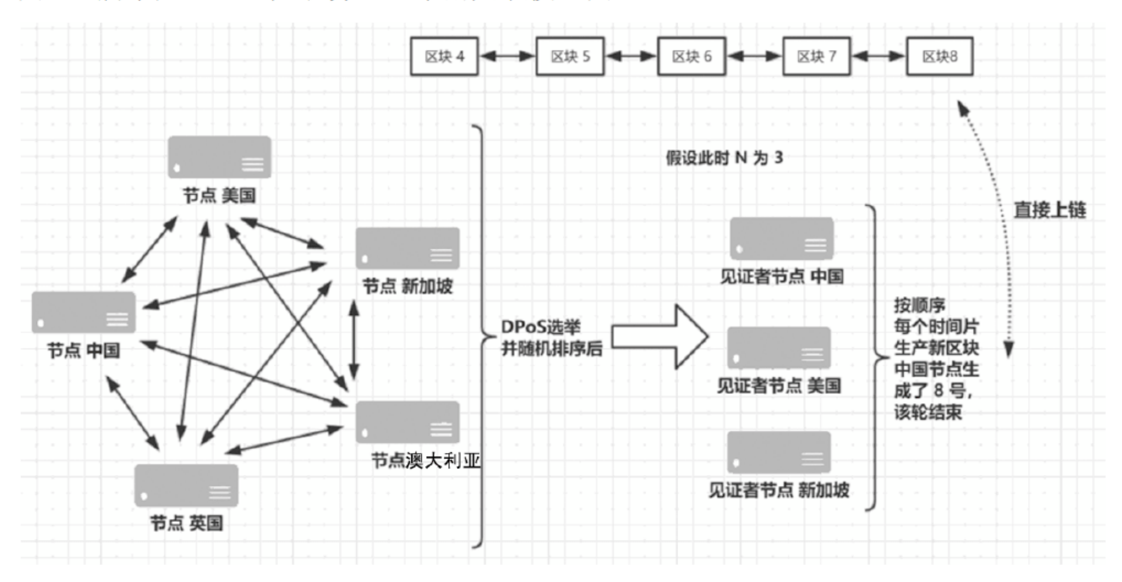



图 1-4　用 DPoS 共识算法选出胜出块

目前，EOS 的超级节点有 21 个，也就是说 N=21，但是拥有 EOS 代币能投票的节点却有很多，那么为什么 N 这么小呢？原因是这样的：虽然 N 代表的是节点们认同的一个能够代表已经足够去中心化的值，但是 N 可以取 23，也可以取 25，或者更大，这些数看起来都足够去中心化了，事实上在 EOS 公链的发展过程中，N 值渐渐地趋向于既要足够去中心化又要让性能跟得上，即处于中间值，以达到平衡性能的同时又满足去中心化。

**提　示**

EOS 的投票事实上还是一种抵押。投票的 EOS 会被抵押成资源，例如 CPU 和网络资源，但是抵押的 EOS 也可以被换回来。把抵押的资源换回 EOS 通常需要 3 天时间。

## 1.3.4　共识算法的编码尝试

本节尝试使用伪代码来实现 3 种共识算法，之所以使用伪代码是为了使读者更容易理解。

### 1.实现 PoW 共识算法

首先，区块链中的各个节点会互相通信以广播新生成的区块。这里既可以用“生成”一词来描述，也可以使用“挖出”一词来描述，表达的意思是一样的，都是指区块的产生。

此时，我们需要使用一个候选区块数组来保存每一个节点广播过来的和自己当前节点生成的区块对象，以及一个全局的区块数组来表示当前公链的区块。区块数组的定义如下：

```
globleBlocks        []Blocks  // 公链区块数组
candidateBlocks     []Blocks  // 候选区块数组
```

假设 Block 结构体内的数据类型如下所示：

```
type Block struct {
    Timestamp   string   // 时间戳，代表该区块的生成时间
    Hash        string   // 这个区块的哈希值
    PrevHash    string   // 这个区块的上一个区块的哈希值
    NodeAddress string   // 生成这个区块的节点地址
    Data        string   // 区块携带的数据
}
```

然后我们需要一个难度系数的变量，例如 difficulty，用来控制 PoW 算法的难度。这个数不一定越大就代表越难，只需要体现出 PoW 算法所描述的工作量难度即可。假设它是整型数，数值越大，计算难度就越大，那么此时 difficulty 系数会处于被随时调节的状态中。在区块链的设计中，例如比特币 BTC 的难度系数就有其动态调节的算法。

这里，我们假设难度系数 difficulty=1。

有了难度系数后，还需要一个专门用来根据 difficulty 校验区块哈希值的函数。我们现在需要假设一种难度的验证算法，假设用哈希值前缀 0（值 0x 后的 0）的个数来和 difficulty 做比较，如果哈希值包含这些前缀 0，那么校验通过。请注意，这是一种很简单的验证算法，且个数很有限，而在比特币公链中，则要复杂得多。

```
func isBlockHashMatchDifficulty(hash string, difficulty int) bool {
    prefix := strings.Repeat("0", difficulty) // 根据难度值生成对应个数的前缀 0
    return strings.HasPrefix(hash, prefix)  //进行前缀 0 个数的比较,包含则返回 true
}
```

现在假设节点启动了一个子协程，一个用来生成区块的方法，并添加到候选区块数组中去，等待校验。下面的这个方法（函数）是用来生成新区块的：

```
// oldBlock 将会从 globleBlocks 中取 len-1 下标的区块
func generateBlock(oldBlock Block, data string)  Block  {
    var newBlock Block
    t  := 秒级别时间戳
    newBlock.Index = oldBlock.Index + 1
    newBlock.Timestamp = t.String()
    newBlock.Data = Data  // 区块的附属数据
    newBlock.PrevHash = oldBlock.Hash  // 区块的父区块的哈希值
    for i := 0; ; i++ {  // 无跳出表达式的 for 循环，代表不断地计算合法的区块
        newBlock.Nonce = hex
        newBlock.Hash = calculateHash(newBlock)
        if isBlockHashMatchDifficulty(newBlock,difficulty) {
            // 自校验一次难度系数，进入到这个 if 里面，证明难度符合，计算出了答案
            candidateBlocks = append(candidateBlocks,newBlock) // 添加到候选区
块
            break
        }
      }
    return newBlock
}
```

假设节点启动了一个子协程且在不断地计算候选区块数组中区块的哈希值，所计算出的哈希值满足难度系数 difficulty 的检验。

```
var resultBlock Block
for block ~ candidateBlocks {
    if isBlockHashMatchDifficulty(block,difficulty ) {
        //请注意，为什么这里又要校验一次，不是在生成的时候校验了一次吗?
        resultBlock = block
        break
    }
    continue
}
// 后续便广播胜出区块，并附带信息，当前这个节点已经确认了。
```

在各个节点确认的过程中，如果达到了所规定的节点数量，那么我们就判断该区块胜出，最终被公链接纳。

最后解答一下伪代码中留下的疑问——为什么还要进行一次校验才广播块呢？因为难度系数 difficulty 是动态改变的，且候选块数组中的 difficulty 不一定就是我们当前的节点所生产的，即使是当前节点生产的，也有可能在生成的时候难度系数已经被出块了，所以在最后广播的时候还需要根据最新的 difficulty 难度系数再做一次校验。

### 2.实现 PoS 共识算法

相对于 PoW，由于 PoS 共识算法没有“挖矿”的概念，且它不是靠计算工作量来进行共识的，体现在代码上也会是另外一种情形。

首先我们依然需要定义一个候选区块数组来保存每一个节点广播过来的和自己当前节点生成的区块对象：

```
candidateBlocks []Blocks   //候选区块数组
```

每个区块结构体有一个变量，用来记录生成这个区块的节点地址。这个变量在 PoW 的伪代码实现中并没发挥作用，但是在 PoS 中却很重要。同样地，和上述 PoW 一样，我们定义如下的区块结构体：

```
type Block struct {
    Timestamp   string   // 时间戳，代表该区块的生成时间
    Hash        string   // 这个区块的哈希值
    PrevHash    string   // 这个区块的上一个区块的哈希值
    NodeAddress string   // 生成这个区块的节点地址
    Data        string   // 区块携带的数据
}
```

其中，NodeAddress 变量用来记录区块的节点地址。

其次，需要有一个子协程，专门负责遍历候选区块数组，并根据区块的节点地址 nodeAddress 获取节点所拥有的代币数量，然后分配股权。

```
stakeRecord []string  // 股权记录数组
for block ~ candidateBlocks {
   coinNum = getCoinBalance(block.NodeAddress)  // 获取节点的代币数量，即股权
   for i ~ coinNum {  // 币有多少，就循环添加多少次
       if stakeRecord.contains(block.NodeAddress) {  // 是否已经包含了
           break // 包含了就不再重复添加
       }
       stakeRecord = append(block.NodeAddress)  // 添加，循环次数越多，股权越多
```

```
    }
}
```

接下来，从 stakeRecord 中选出一个竞选胜利者。这个概率和上面的 coinNum 有关，coinNum 越大就越有机会。为什么呢？因为它的统计方式是用 coinNum 作为循环界限，然后对应添加 coinNum 次的 nodeAddress，所以 coinNum 越大，这个 nodeAddress 就被添加得越多，后面节点能被选上出块的概率也就越大。

这里还要解答一个疑点，为什么已经包含了的就不再重复添加，因为当前的候选区块数组 candidateBlocks 中可能含有同一个节点中的多个区块，而每一个节点中的股权只需要统计一次，即 coinNum 只需要循环一次即可。如果是多次循环，就会造成不公平，因为会造成多次添加。

在股权被分配好后，接下来准备选出节点胜利者。选择的方式也是使用算法，在这个例子中我们依然采取最简单的随机数的形式进行选择。注意，切勿被这样的方式限制了思维，这个选择算法是可以自定义的，因而可以是更加复杂的算法。

```
index := randInt()  // 得出一个整型随机数
winner := stakeRecord[index]  // 取出胜利者节点的地址
```

在最后的步骤中，就能根据这个 winner 去所有候选区块中选出节点地址和它一样的区块，这个区块就是胜利区块，将会被广播出去。

```
var resultBlock Block
for block ~ candidateBlocks {
    if block.NodeAddress == winner {
        resultBlock = block // 添加
        break
    }
}
// 广播出去，等一定数量的节点同步后，就会被公链接纳
```

以上是一个很简单的 PoS 算法机制的代码实现，仅单纯地根据持币数量来进行股权分配。事实上，事情往往是比较复杂的。设想股权的分配不仅只和代币数量有关，例如以太坊设想的 PoS 共识算法的实现中加入了币龄，情况又会如何呢？这时在候选成功后，以太坊会扣除币龄。作为开发者应当理解 PoS 的精髓——其算法的实现往往会衍生出各种各样的变种，只有了解了这一点，才能在开发自己的公有链时随心而行。

### 3. 实现 DPoS 共识算法

DPoS 的伪代码实现可以理解为 PoS 的升级版，之前例子中相同的数据结构体的定义这里不再重复。

首先定义好见证者节点的结构体：

```
type WitnessNode struct {
  name       string  // 名称
  Address    string  // 节点地址
  votes      int     // 当前的票数，见证者是投票产生的
}
```

然后我们用一个由各个见证者节点组成的数组代表这一批见证者节点，往后的随机排序操作也将会在这个数组中进行。

```
var WitnessList []WitnessNode  // 见证者节点
```

现在我们需要准备一个专门用来对 WitnessList 进行随机排序的方法，这个方法须依赖某种算法对 WitnessList 进行排序，具体算法可以自定义，但要根据不同的业务需求而定。

下面我们依然以一个最简单的随机数排序为例。

```
func SortWitnessList()  {
    if NeedRestVotes() {    // 判断是否需要重新投票
        for witness ~ WitnessList  {
            witness.votes = rand.Intn(100)  // 进行投票
        }
    }
    SortByVotes()  // 根据票数排序
}
```

上面 NeedRestVotes()的作用是判断是否需要重新投票选出见证者节点，对应 DPoS 算法描述中的每过一个周期就开始重新排名，在这个阶段还需要进行节点的剔除，例如剔除一些坏节点。

这里以检查坏节点为例，因为坏节点的检查时刻在进行，所以我们可以用一个子协程 (Go 语言中，协程是一种轻量级线程，为了更加贴切，下面的 Go 代码中统称线程为协程) 来专门检查坏节点。

```
func CheckBadNode(){
    for witness ~ WitnessList {
        if isBadNode(witness) {   // 判断是否为坏节点
            WitnessList.remove(witness)  // 是的话就移出它
        }
    }
}
```

同时，还要不断地检测是否有新的见证者节点被投票选出，是的话，就要添加这个节点的信息进入到见证者节点数组中。同时要对见证者节点总数的数量进行 N 值限制。

```
func CheckNewWitnessNode(){
    for  {
         if WitnessList.size() < N {  // 判断是否超过 N 值
           newWitness := isNewNodeComing()  // 检查是否有新的见证者节点到来
           if newWitness != nil {
               WitnessList = append(WitnessList,newWitness)  // 添加
           }
           time.sleep(50ms)  // 延时一段时间，进行下一轮的检测
         }
     }
}
```

最后我们使用出块函数从 WitnessList 见证者列表中从上到下逐个找出出块节点，进行出块，并检测当前轮到的节点是否出块超时，超时就轮到下一个，以此类推，对应 DPoS 共识算法的伪代码如下：

```
func MakeBlock()  {
    SortWitnessList()  // 开始的时候，进行一次排序
    for { // 无跳出表达式的 for 循环，代表内部不断地计算合法的区块
        witness := getWitnessByIndex(WitnessList)// 从上到下获取
```

```
            if witness == nil {
                break  // 所有见证者出块都出了问题
            }
            block,timeOut := generateBlock(witness)  // 传入该见证者节点
            if timeOut {   // 是否超时
                continue   // 超时就轮到下一个
            }
            // 广播 block 块出去，然后结束该轮，等待下一次开始
            break
        }
    }
```

在广播块出去后，其他见证者接收到了广播，会对这个区块进行签名见证，当达到了某个我们所设定的认为见证已经足够了的值时，那么这个区块就被确认了。

从上述伪代码发现，传统的 DPoS 算法直接应用的时候存在如下问题：每个见证者节点都是循环着使用别的节点信息去生成块，而不是使用自己节点的信息。假设见证者节点 A、B、C 在一轮的出块顺序中，节点 C 排在第一位，且在节点 A 和 B 中使用节点 C 所生成的区块都没有出差错，那么节点 C 就会在节点 A 和 B 中都生成一次，由于时间戳不一样，导致该块的哈希不确定，但最终只能有一个块被选上，这样就导致了算力浪费。当然，也有可能不止节点 A 和 B，还可能有更多的节点都使用节点 C 生成了区块，那么结果就是更多的认证者节点生成了一个节点的块，去广播。

要解决上述问题，可以使用 EOS 的做法：EOS 通过见证者节点信息注册，使得每个节点都知道所有见证者节点的信息，同时被注册的节点都必须是满足投票条件的。

当每个见证节点都有了所有见证者节点的信息后，在每一次的最终块出现后，都会使用特定的算法对节点列表进行排序。如此，当需要出块的时候，节点会根据区块链中最后一个区块的时间来参与到某些计算中，得出当前应该出块的见证者节点在列表中的下标，然后判断这个节点是不是自己。不是的话，就会让自己延迟（delay）一定的时间，然后重复上面的步骤，这个延迟的时间就是 DPoS 中的出块超时，然后会自动轮到下一个见证者节点。如果是自己就出块，出块后就广播出去，等到 2/3 的见证者节点都签名确认了，那么这个块就是最终有效的。所以，EOS 中的 DPoS 并不是传统示范代码中的那样，一个节点循环着生成含有别的节点的信息的块。

EOS 的要点是，每个见证者节点的自身代码对所有见证者节点的排序是不一样的，各节点存在同时出块的可能，但其提供了 2/3 见证者节点都签名确认这一环节，即谁最快被 2/3 的见证者节点确认，谁才是最终有效的，解决了多个节点同时生成一个节点块的问题。

## 1.4 链的分叉

上一节我们介绍了 3 种常见的共识算法：PoW、PoS、DPoS。虽然它们都让区块链中的各个节点在一定程度上做到了共识，但是也会产生不可避免的问题——链的分叉。本节我们来认识一下什么是链的分叉。

我们知道，区块链中的每一个区块在节点中被生成后都会通过 P2P 网络广播到其他节点中去，

这些节点都是同一类节点，它们组成一类节点的节点网络。例如，比特币公链的节点就是比特币公链的，以太坊就是以太坊的，而不能是比特币公链的节点广播区块到以太坊的节点中去。

广播后的区块在到达了其他节点后，其他节点要对该区块进行操作，例如进行签名操作。然后，各个节点在广播给它的一批又一批的区块和它自己所产生的区块中做出抉择，即选出一个获胜的区块。

然而，共识算法仍然无法保证不出现确认冲突的问题，例如比特币中的 PoW 共识算法依赖谁算出合法哈希且谁算得快来抉择最终选谁。事实上，即使我们产生区块的时间戳精确到毫秒级，依然会出现同时算出哈希值的多个节点（至少有两个节点），例如节点 A 和节点 B 同时算出了合法的哈希值，产生了区块 1，广播出去了。节点 C 也陆续收到了节点 A 和节点 B 的区块 1，但是节点 C 首先收到的是节点 A 的区块 1，此时虽然节点 B 的区块 1 也合法，但是也不采纳了。同时，节点 D 也收到了节点 A 和节点 B 的区块 1，但是节点 D 先收到节点 B 的区块 1，为什么？因为节点 D 的网络路由距离节点 B 的网络近，离节点 A 的网络远，那么节点 D 就会先采纳节点 B 的区块 1 而不采纳节点 A 的区块 1。此时，链就分叉了，如图 1-5 所示。

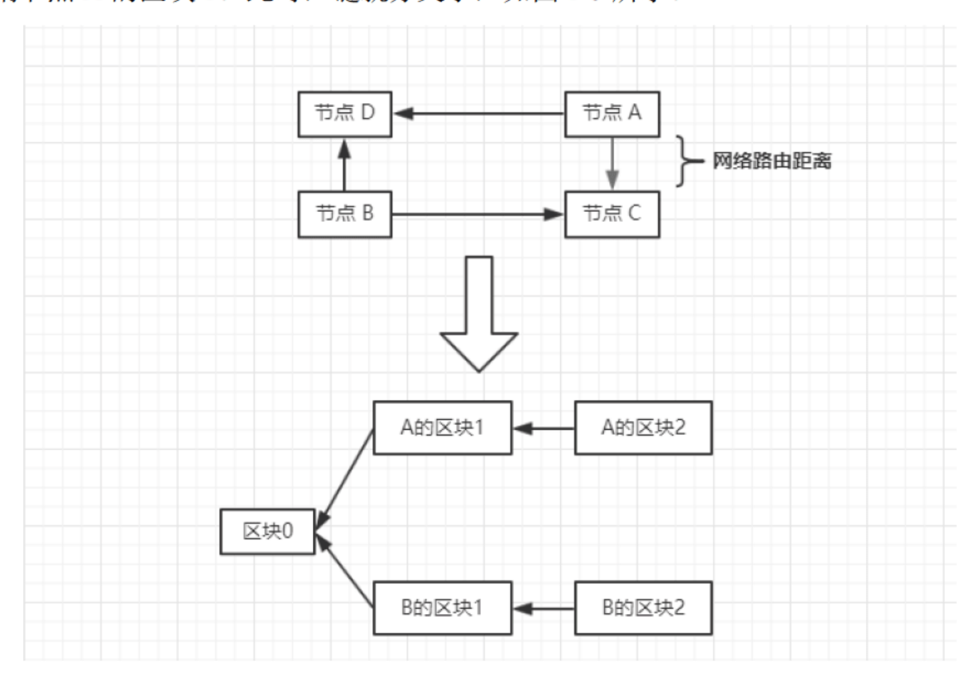



图 1-5　节点广播块时路由距离的影响

我们把这个例子定为情况①。这是一个很简单的分叉模型。

链的分叉主要有以下两种情况：

（1）硬分叉。一旦出现，最后的结果是一分为二，专业的说法是：旧节点无法认可新节点产生的区块，称为硬分叉。

（2）软分叉。一旦出现，最后的结果是能掰正的，专业的说法是：旧节点能够认可新节点产生的区块，称为软分叉。

## 1.4.1 软分叉

情况①就是软分叉的一种。当有两个或多个节点同时挖出了同区块号码的一个区块，然后它们同时广播信息出去，假设一个是节点 A，而另一个是节点 B，那么距离节点 A 比较近的节点，还没收到其他节点的消息就先收到了节点 A 的信息，并开始确认节点 A 所挖出的这个区块的信息，随后把节点 A 挖出的这个区块加入自己所在的公链中；同理，距离节点 B 比较近的节点也会先处理节点 B 挖出的区块信息，并把节点 B 挖出的这个区块加入自己所在的公链中，如图 1-6 所示。

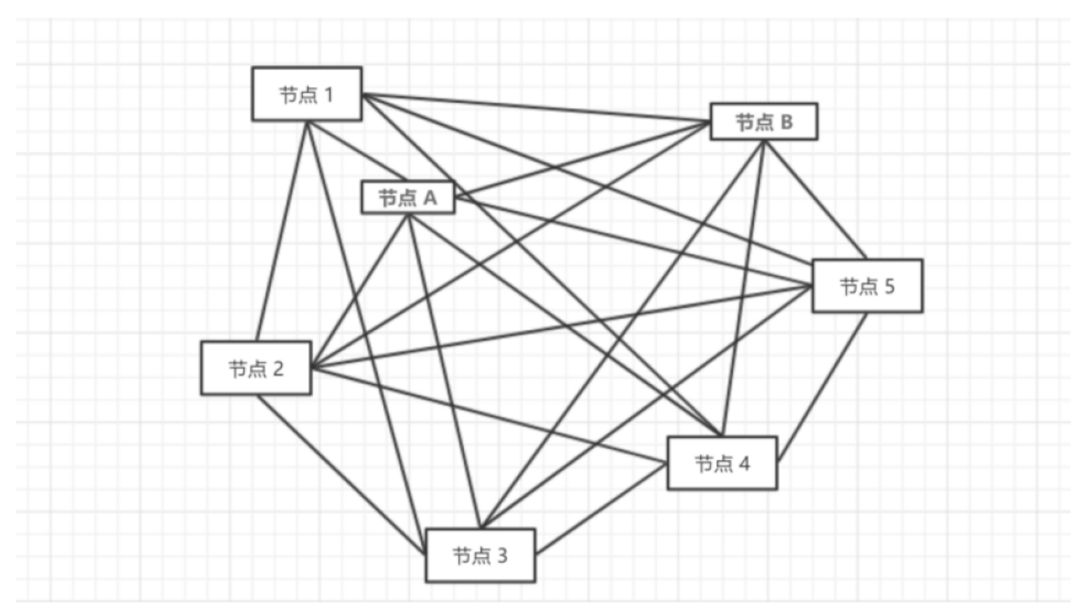



图 1-6　节点网络

情况①的链分叉是各个节点在使用了同样的共识算法下导致的分叉，如图 1-7 所示。

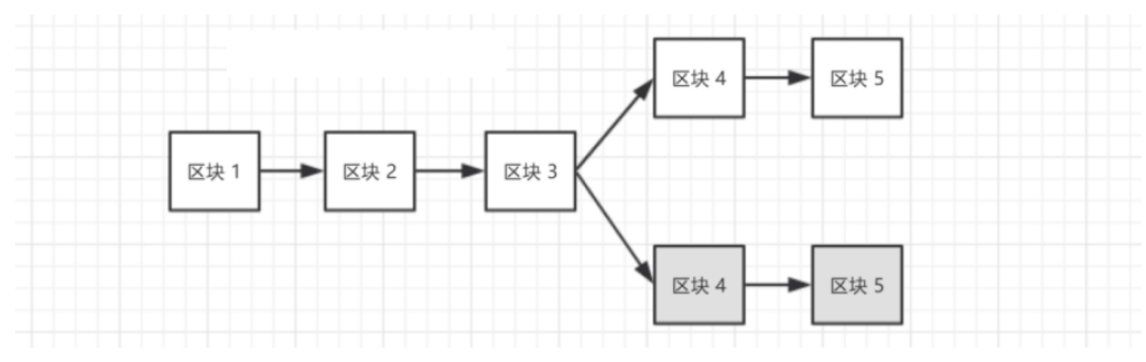



图 1-7　链的分叉

出现这种情况，矿工是比较容易自我纠正的。由于节点网络的整体解题能力和矿工的数量成正比，因此链的增长速度也是不一样的，在一段时间之后，总有一条链的长度会超过另一条。当矿工发现全网有一条更长的链时，他就会抛弃当前的链，把新的更长的链复制过来，在这条链的基础上继续挖矿。所有矿工都这样操作，这条链就成为了主链，分叉出来的那个链便会被抛弃掉。但是，并不是所有的分叉都能被自动纠正，这点请注意，具体会在后面的章节中进行说明。

这种软分叉的自我纠正机制也是区块链的一个重要特点，就是最优链的选择。注意，不同的

公链，它们的最优链选择算法并不一样。常见的选择机制有：

（1）最长链机制。整条区块链以最长链为主，且各个节点根据长链不断同步，目前比特币公链采用的就是这种机制。

（2）其他链选择机制。例如，以太坊的“Ghost 协议”机制，将会在“以太坊 Ghost 协议”一节中进行详细介绍。

现在我们可以解答在“DPoS 算法”一节中的问题——为什么 DPoS 共识算法下的 N 必须取奇数的原因：取奇数是为了避免在 DPoS 共识算法下出现链的分叉。因为在 DPoS 中，区块在广播后必须被见证者节点签名认证，在达到 2/3 数目的见证者节点签名后就宣布该块胜出。在奇数的情况下，是永远不可能出现对半的情况的，例如 10 个节点签名了区块 A、另外 10 个签名了区块 B。所以，奇数的见证者节点数，即 N，很好地避免了链的分叉问题。

注意，链的分叉在不同的共识算法中对应着不同的解决策略。

图 1-8 所示为链分叉后最长链自我纠正机制的模型图。

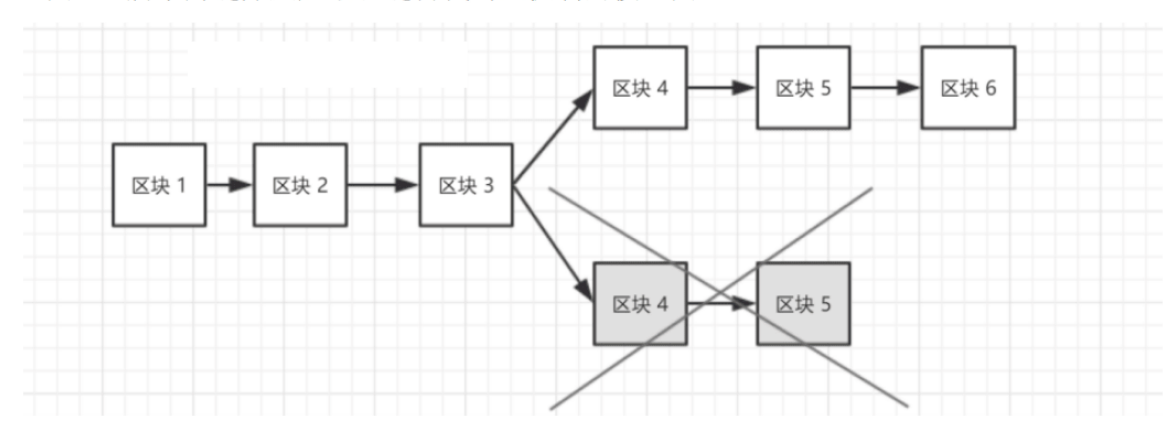



图 1-8　最长链自我纠正机制

软分叉除了上面的情况①之外，还有另外一种情况，因共识规则改变，旧节点能够识别新节点产生的区块，但旧的区块不能被新节点接受，这种情况又分为下面的两种形式：

- 新节点全网算力大于等于 51%。
- 新节点全网算力小于 50%。

这种软分叉不一定能由节点自我纠正，解决办法是必须依赖人力升级节点到同一版本。

（1）当新节点的全网算力大于等于 51%时，无论旧节点升级不升级，最长的链最终都是由全部新节点生成的区块所组成的链，而且这条最长链都是新旧节点双方认为合法的一条，原因参考上面讲解的最长链复制机制：

- 旧的能接收新的，在分叉点之后的区块掺杂着：
  - 旧节点的区块。
  - 新节点的区块。
- 新的不能接收旧的，但最终新的总比旧的长。

（2）当新节点的全网算力小于 50%时，最终不能通过短的复制长的达到统一，结果是硬分叉。

原因如下：

- 旧节点比新节点的链要长。
- 新的总是不能接受旧的，不会去复制一条含有自己不能接受的区块的链。

### 1.4.2 硬分叉

如果区块链软件的共识规则被改变，并且这种规则改变无法向前兼容，旧节点无法认可新节点产生的区块，且旧节点偏偏就是不进行软件层面的升级，那么该分叉将导致链一分为二。

分叉点后的链互不影响，节点在"站队不同派别"后也不会再互相广播区块信息。新节点和旧节点都开始在不同的区块链上运行（挖矿、交易、验证等）。

举个简单的例子，如果节点版本 1.0 所接收的区块结构字段是 10 个，1 年后发布节点 2.0 版本，2.0 兼容 1.0，但是 1.0 的不能接受 2.0 版本中多出的字段，即出现了硬分叉。

硬分叉的过程如下：

（1）开发者发布新的节点代码，新的节点代码改变了区块链的共识规则且不被旧的兼容，于是节点程序出现了分叉（Software Fork）。

（2）区块链网络部分节点开始运行新的节点代码，在新规则下产生的交易与区块将被旧节点拒绝，旧节点开始短暂地断开与这些发送被自己拒绝的交易与区块新节点的连接，于是整个区块链网络出现了分叉（Network Fork）。

（3）新节点的矿工开始基于新规则挖矿，旧节点的矿工依然用旧的规则，不同的矿工算力导致出现分叉（Mining Fork）。

最终，整个区块链出现了分叉（Chain Fork）。

**实例：**

"2017 年 8 月 1 号，Bitcoin Cash（BCH）区块链成功地在区块高度 478559 与主链分离。这一新的加密货币默认区块大小为 8MB，并且可以实现区块容量的动态调整。由于旧节点只认可小于 1MB 的区块，所以运行 BCH 客户端节点产生的区块无法向前兼容，将被旧节点拒绝，最后运行不同客户端的矿工将会长期运行在两条不同的区块链上（BTC 和 BCH）。"

### 1.4.3 常见的分叉情况

本节我们从正常的区块的生成流程开始，使用 DPoS 共识算法模式，分析可能会出现分叉的各种情况。

在正常操作模式下，区块生产者每 3 秒钟轮流生成一个区块（见图 1-9）。假设没有节点错过自己的轮次，后续便进入到选出胜出区块的步骤。注意，区块生产者在被调度轮次之外的任何时间段出块都是无效的。

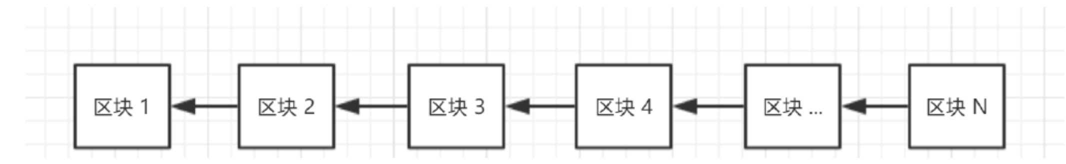



图 1-9　正常形态的链

下面假定节点网络中共有 5 个节点，分别是 A、B、C、A1、B1。

### 1.少数节点分叉

这是最常见最简单的一种软分叉，是由不超过节点总数 1/3 的恶意节点创建或因区块确认时间差导致的分叉现象。在这种情况下，假设少数节点分叉每 6 秒只能产生 2 个块，而多数节点分叉每 6 秒可以产生 3 个块（因为多数节点在同步速度上是比单节点向别的节点确认它的块的时间要短），这样诚实的 2/3 多数节点维护的链将永远比分叉的链更长。

如图 1-10 所示，每个块中的字母代表是哪个节点产生的。

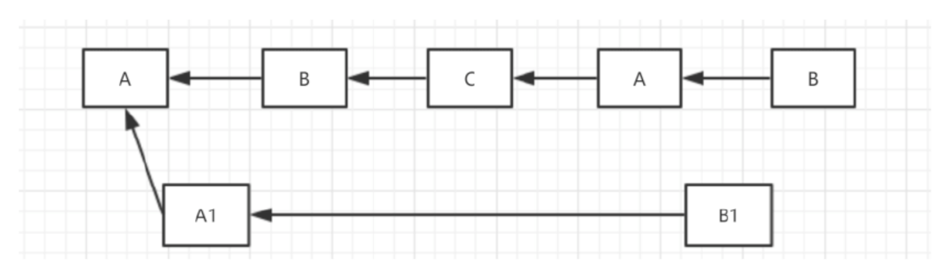



图 1-10　分叉链含有较少的块

### 2.网络分片化分叉

当节点网络中部分节点由于网络波动或其他原因导致自己与全网节点的网络断开了连接，即会出现网络分片化分叉，如图 1-11 所示。

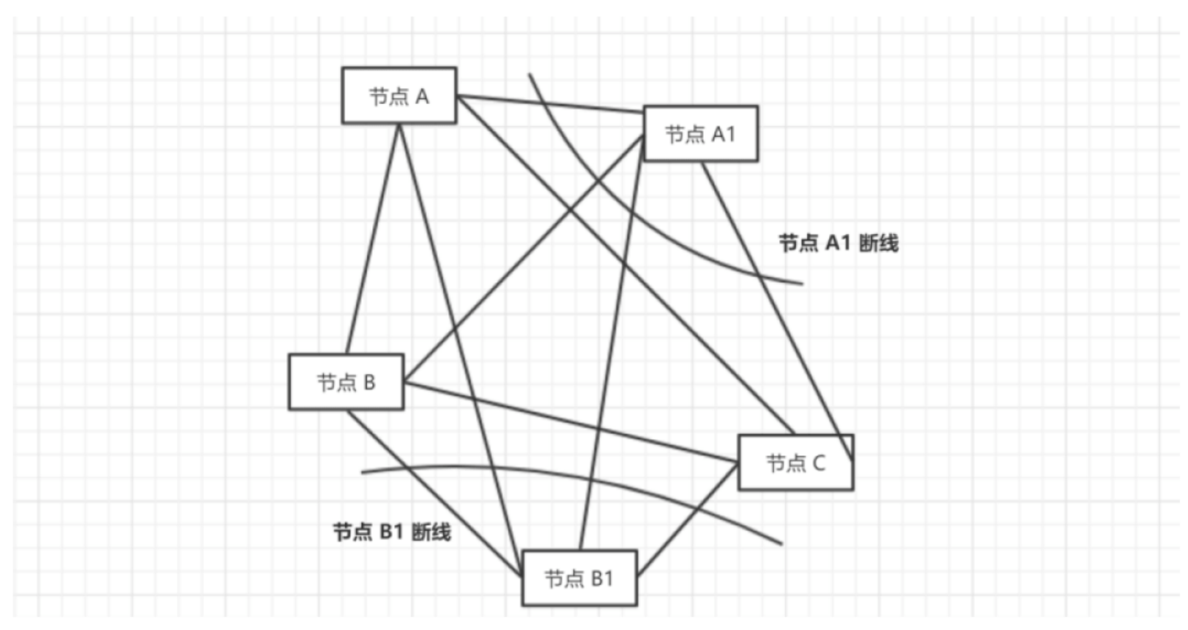



图 1-11　Node 节点网络部分节点掉线

节点 A1 和节点 B1 断开了与主网的连接，此时它们的块生产的部分代码依然还在运行着，这样生成的块就只能归纳到本地所维护的公链区块数组中，节点网络被分成了 3 部分。请注意，无论如何分叉，根据链选择算法，其中总会有一条最优链，所以最终依然只有一条主链。如图 1-12 所示，每个块中的字母代表是哪个节点产生的。

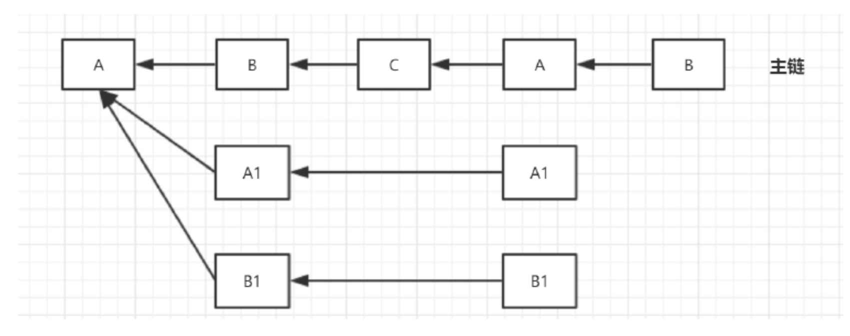



图 1-12　多条分叉链

如果断线的节点网络恢复后，重新连接上了主网，那么它会自动进行最优链的复制，最终的结果也是只有一条最优链。

如果分叉节点永远地脱离了主网，结果就会造成硬分叉。如果脱离出的节点数目不多，那么这些脱离的节点就会变成私有节点或组成一个联盟链网络。

#### 3.多数节点舞弊分叉

所谓舞弊，就是我们所理解的作弊的意思。节点作弊是指节点不遵循共识规则或做了一些非法操作，例如尝试修改块。这种情况下所导致的链分叉称为舞弊分叉。这类分叉和网络分片化的模型图很类似，不同点在于，舞弊是节点们都还在同一个主网中产生的。

因此，舞弊导致的分叉不会太久，最终还是会被诚实节点的最优链纠正过来。假设节点 A1 和节点 B1 是舞弊者，A、B、C 是遵守规则的节点，其模型图如图 1-13 所示。

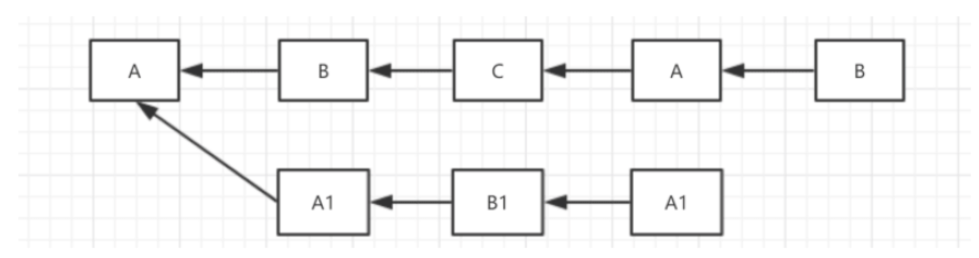



图 1-13　节点模型图

以上是目前常见的链分叉情况。实际中还有很多其他复杂的情况，但是无论何种情形，只要不是硬分叉，最终都是可以被纠正的。纠正的主要手段有以下两种：

（1）最优链复制同步。

（2）人工通过技术手段纠正。

### 1.4.4　PoW 共识机制的 51%算力攻击

51%算力攻击目前仅在“PoW”共识机制中存在，因为“PoW”共识机制依赖算力计算获胜，

也就是谁算得快，谁的胜率就高。在使用了“PoW”共识机制的区块链网络中，我们称参与计算哈希的所有计算机资源为算力，那么全网络的算力就是 100%，当超过 51%的算力掌握在同一阵营中时，这个阵营的计算哈希胜出的概率将会大幅提高。

为什么是 51%？50.1%不行吗？当然也是可以的，之所以取 51%是为了取一个最接近 50%，且比 50%大的整数百分比，这样当算力值达到 51%后的效果将会比 50.1%的计算效果更明显。举个例子，如果诚实节点的算力值是 50.1%，那么坏节点的算力值就是 49.9%。两者的差距不算太大，这样容易导致最终的区块竞争你来我往、长期不分上下。

如果算力资源分散，不是高度集中的，那么整个区块链网络是可信的。然而，当算力资源集中于某一阵营的时候，算力的拥有者就能使用算力资源去逆转区块，导致区块链分叉严重，如下面的例子。

假设图 1-14 是一条区块链目前的状态。一个攻击者想要逆转区块 8 中的一笔交易，他就会从区块 7 后面引入一个分叉来使区块 8 变得无效，在分叉块中设置给某个地址几百或者几千个 BTC。不过，由于比特币公链的最长链规则的限制，所有的诚实节点都会遵循最长链规则，将新产生出来的区块链接在最长链的尾部，从而避免攻击者得逞。

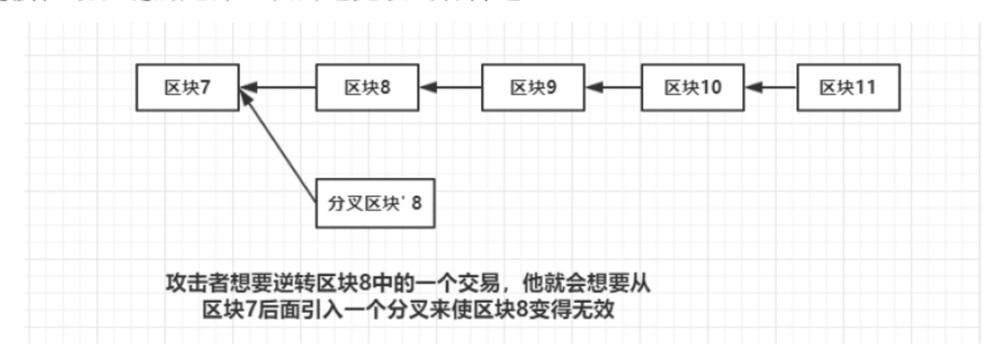



图 1-14　某条区块链的状态

当系统出块率比较低且块大小较小时，网络延迟相对于出块时间来讲是比较小的，这样诚实的节点所产生的区块基本上就是顺序的。只要诚实节点的总算力超过 50%，攻击者就不能够使它们自己产生的链成为最长链。然而，当诚实节点的总算力不及坏节点的算力时，即坏节点算力总和超过了 51%，最长链机制将会被坏节点利用，因为此时坏节点的出块速度整体比诚实节点快，获胜率高，这样坏节点产生的区块将会形成最长链。

此外，如果出块率很高，会使得区块产生的时间和区块在网络上传播的延迟相对变得较小，这样一个新块在产生以后还来不及传播到全网就会有其他的节点产生别的新块，互相竞争剧烈，导致链上分叉情况严重。虽然最终只会有一条最长链，但是出块率越高，块大小越大，分叉的情况就会越严重，最终区块链就会发展成有很多分叉的样子，如图 1-15 所示。

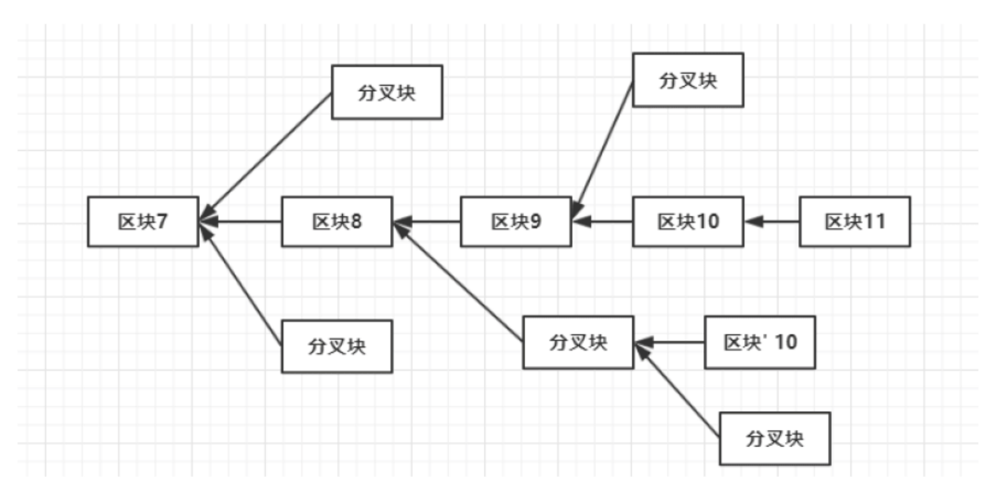



图 1-15　复杂的分叉情况

基于比特币公链来看（以太坊公链中分叉块有其他处理），大量的分叉会带来两个问题：

（1）浪费了网络资源和计算资源，大部分分叉块无效，因为只有最优链中的区块才被认为是有效的。

（2）危害了安全性，整个区块链里的最优链变短了，算力分散在不同的分叉链中，这使得攻击者只需要少于 51%的算力就可以产生出恶意的最优链。就好比有 3 个阵营，A 阵营有 30%算力，B 阵营有 32%算力，C 阵营有 38%算力，算力以 3 大阵营分散在 A、B、C 上，如果 A、B、C 各自搞分叉，那么最终 C 就可以以低于 51%的算力（38%的算力）达到制造恶意最优链的目的。

# 1.5　小　结

本章主要介绍了区块链的经典知识点，包含区块链的定义、链的分类、共识算法与伪代码的实现。着重讲解了链分叉的定义和分叉的两种类型，硬分叉与软分叉的定义及其产生的原因，并讲解了常见的 3 种软分叉。最后结合“PoW”共识机制介绍了区块链中著名的 51%算力攻击。

# 第 2 章

# 以太坊基础知识准备

在上一章中，我们介绍了区块链的基础知识，本章将开始介绍以太坊 DApp 开发十分重要的预备知识。如果你想基于以太坊开发应用，请务必掌握本章内容。

## 2.1　什么是以太坊

以太坊其实就是区块链的一种应用，是一条公链，包含但不限于“区块链”所具有的技术特点。

区块链是一个整体的名词，我们可以根据区块链技术开发出很多公链或者私链，再给这些链一个名称，例如使用“以太坊”这个名称。区块链和以太坊的关系如图 2-1 所示。

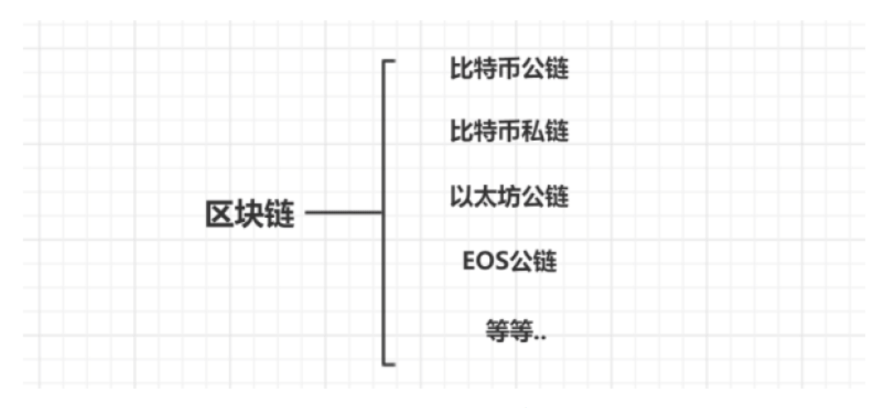



图 2-1　链的分类

公链就是最多节点所共同维护的链；私链是节点比较少的链，一般是一个节点，任何人都可以在自己的电脑上进行私链的部署，只需要下载一份公链的代码，在自己的本地机器上跑起来即可。

目前行业内将区块链应用以版本的形式进行了划分，每个版本有其对应的代表性应用，其中以太坊公链被公认代表了区块链的 2.0 版本，具体如下：

（1）区块链 1.0，代表者是比特币公链，不具备智能合约功能，具备区块链的其他技术模块，是第一条支持电子货币转账的完整区块链公链。

（2）区块链 2.0，代表者是以太坊公链，技术模块方面比比特币公链多出智能合约等创新的功能，其共识机制正在从“PoW”向“PoS”过渡，但是直到现在，以太坊最新版本的共识机制使用的依然是“PoW”，虽然和比特币一样是“PoW”，但是以太坊的性能要比比特币公链高，最主要的原因就在于“PoW”算法的改进以及最优链的判断方法不同。

（3）区块链 3.0，主要目标是实现高性能、大吞吐量，代表者是“EOS”柚子公链，具备智能合约功能，共识算法是“DPoS”，现在正在向“BFT-DPoS”方向发展。

此外，还有一些区块链框架应用，它们不是公链。例如，IBM 公司的“HyperLedger”开源项目就是一个具备技术模块插件化功能的区块链框架，可以使用它来自定义开发公链或联盟链应用。

# 2.2 以太坊的架构

整个以太坊的技术栈可分为应用层、网络层、合约层、共识层、激励层和数据层，共 6 层，如图 2-2 所示。

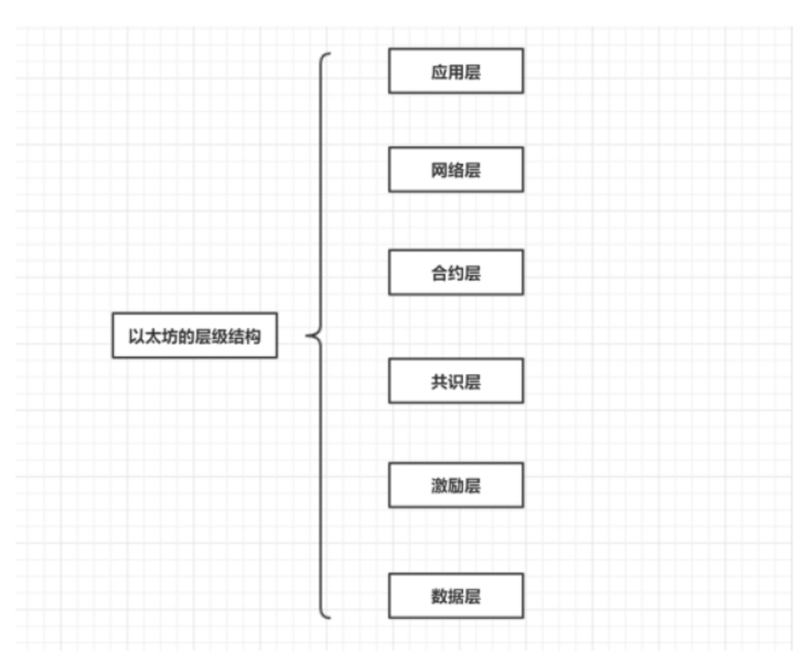



图 2-2　以太坊层级结构

每一个层级对应不同的功能。

- 应用层：主要是基于以太坊公链衍生出的应用，例如各种 DApp 应用、Geth 控制台、Web3.js 接口库以及 Remix 合约编写软件和 Mist 钱包软件等。
- 网络层：主要是以太坊的点对点通信和“RPC”接口服务。
- 合约层：某些公链不具备这一层，例如比特币就没有合约层，以太坊的合约层主要是基于智

能合约虚拟机“EVM”的智能合约模块。

- 共识层：主要是节点使用的共识机制。
- 激励层：主要体现在节点的挖矿奖励。挖出胜出区块的节点或打包了叔块的区块所对应的节点，矿工会获得规则所设定的 ETH 奖励。
- 数据层：用于整体的数据管理，包含但不限于区块数据、交易数据、事件数据以及“levelDB”存储技术模块等。

以太坊的技术细分架构如图 2-3 所示，从上到下，越底部代表越底层。

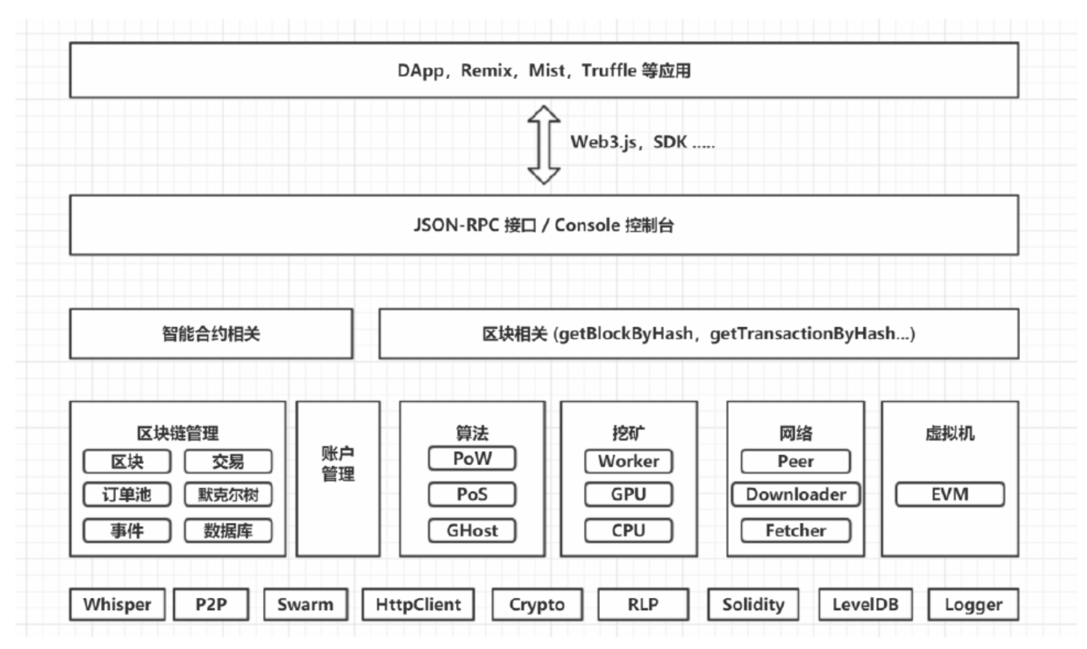



图 2-3　以太坊技术架构图

应用通过 Web3.js 或其他版本的以太坊接口访问代码，来访问以太坊的“RPC”接口获取对应的数据。接口分为与智能合约相关和与区块相关，共两个部分。

Whisper 是 P2P 通信模块中的协议，节点间的点对点通信消息都经过它转发，所转发的消息都经过加密传输，如图 2-4 所示。

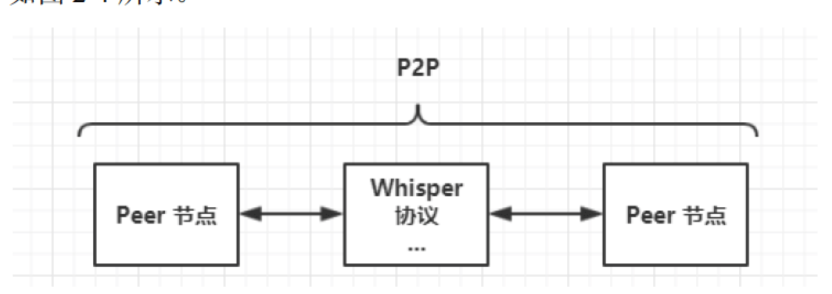



图 2-4　Whisper 协议

Swarm 是以太坊实现的类似于 IPFS 的分布式文件存储系统，在 P2P 模块中结合 Whisper 协议使用。

HttpClient 是 HTTP 服务请求方法的实现模块。

Crypto 是以太坊的加密模块，内部包含 sha3、secp256k1 等加密算法。

RLP 是以太坊所使用的一种数据编码方式，包含数据的序列化与反序列化。关于数据编码方式，除我们常见的方式之外，还有 base16、base32 和 base64 等。

Solidity 是以太坊智能合约的计算机编程语言，由它编写智能合约，使用时由 EVM 虚拟机载入字节码运行。

LevelDB 是以太坊所使用的键值对数据库。区块与交易的数据都采用该数据库存储。此外，在以太坊中，作为键（Key）的一般是数据的“Hash”值，而值（Value）则是数据的“RLP”编码。

Logger 是以太坊的日志模块，主要包含两类日志：一类是智能合约中的事件（Event）日志，该类日志被存储到区块链中，可以通过调用相关的“RPC”接口获取；另一类是代码级别的运行日志，这类日志会被保存为本地的日志文件。

# 2.3 什么是 DApp

## 2.3.1 DApp 概述

DApp 的英文全称是“Decentralized Application”，对应的中文解释是：去中心化应用，又称分布式应用。

关于分布式应用可分为传统的 DApp 和区块链 DApp，下面我们看一下这类分布式应用的不同。

### 1．传统的分布式应用

在区块链出现之前，DApp 已经存在了，我们可把这种 DApp 称为传统的分布式应用。我们以所熟悉的 C/S（Client/Server，客户端/服务器端，亦称为客户机/服务器）结构来看一下这种分布式应用的特点。我们知道，一个 Server 是可以服务于多个 Client 的，如果不考虑各个 Server 之间的通信，那么这种一对多的形式可看作如图 2-5 所示的简单交互形式。

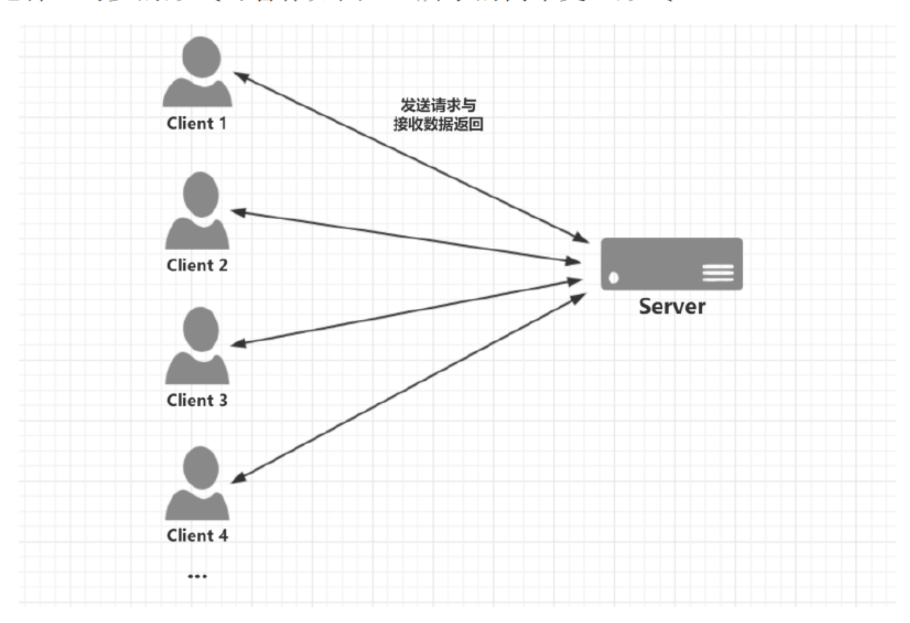



图 2-5　客户端和服务器端交互的示意图

在这种 C/S 结构中，相同功能的 Server 允许有多个，它们可以被放置在不同的地方，如果多个 Server 之间可以相互通信，我们就称 Server 部分为分布式集群，如图 2-5 所示。

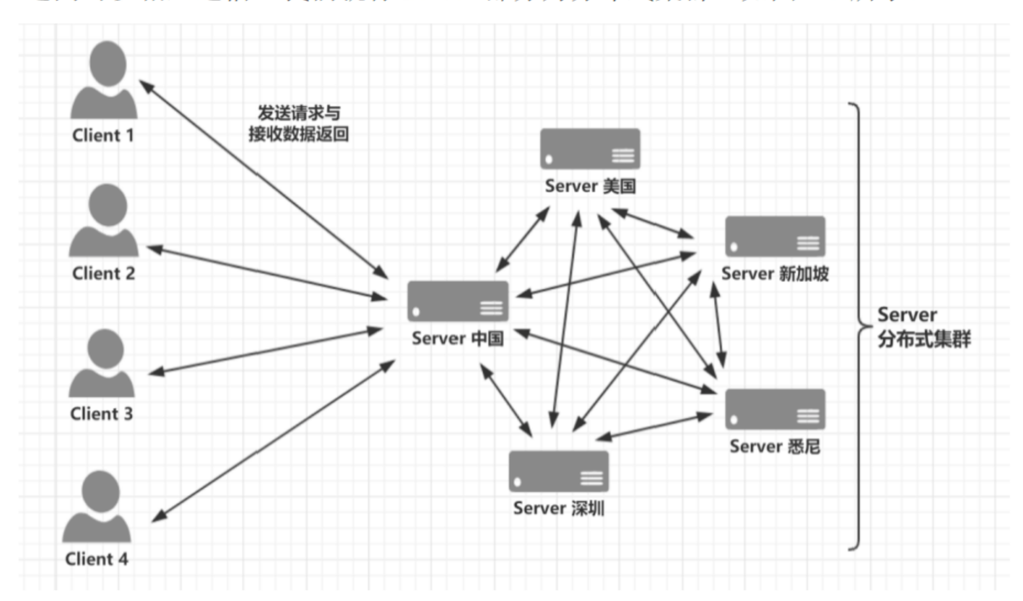



图 2-6　分布式集群的大概模型

图 2-6 这种 Server 分布式集群只是一个大概的模型，事实上还会有其他中间件穿插其间，如图 2-7 所示。在这类传统的分布式应用中，往往是多个 Server 与同一个数据源交互，即多个 Server 进行数据的读写操作。注意，这是一个重要的差异，即传统的分布式应用 DApp，它的数据存储源总是相同的，即使数据源做了分布式集群，也依然不是去中心化的，同时，系统管理员也可以访问数据源。

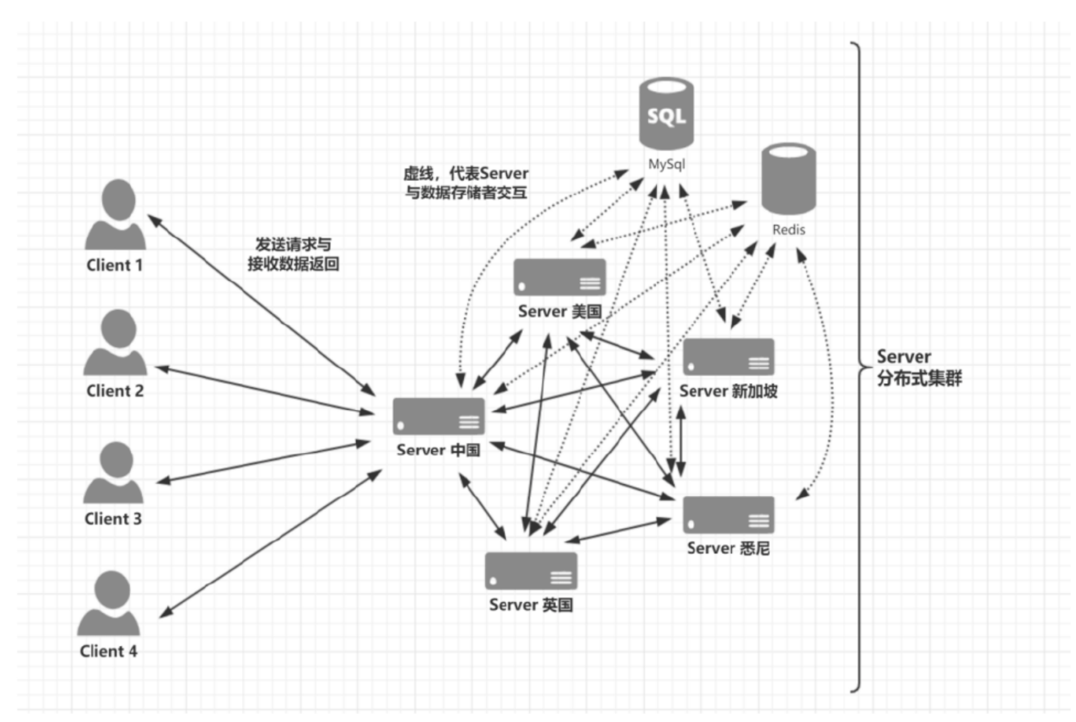



图 2-7　中间件穿插其间的分布式集群

### 2. 区块链去中心化分布式应用

区块链去中心化分布式应用 DApp 与传统的分布式应用 DApp 的最大不同点在于，前者是完全去中心化的，特别是数据存储部分。在区块链这种分布式应用中，Server 被重新命名为节点，名称改变了，但其本质没变，依然是为 Client 提供服务的，只是每个节点由不同的组织管理，并对应有自己的数据存储区域。

去中心化 DApp 每个节点都有自己的数据存储地，而且节点之间可以彼此相互通信却又不依赖其他节点（因为互不信任），例如 A 节点无法直接访问 B 节点的数据库。在这种互不信任的体系中，各个节点通过共同遵循共识算法来达到数据同步的目的，同时各个节点之间又维护了一条区块链。它们的交互形式大致如图 2-8 所示。

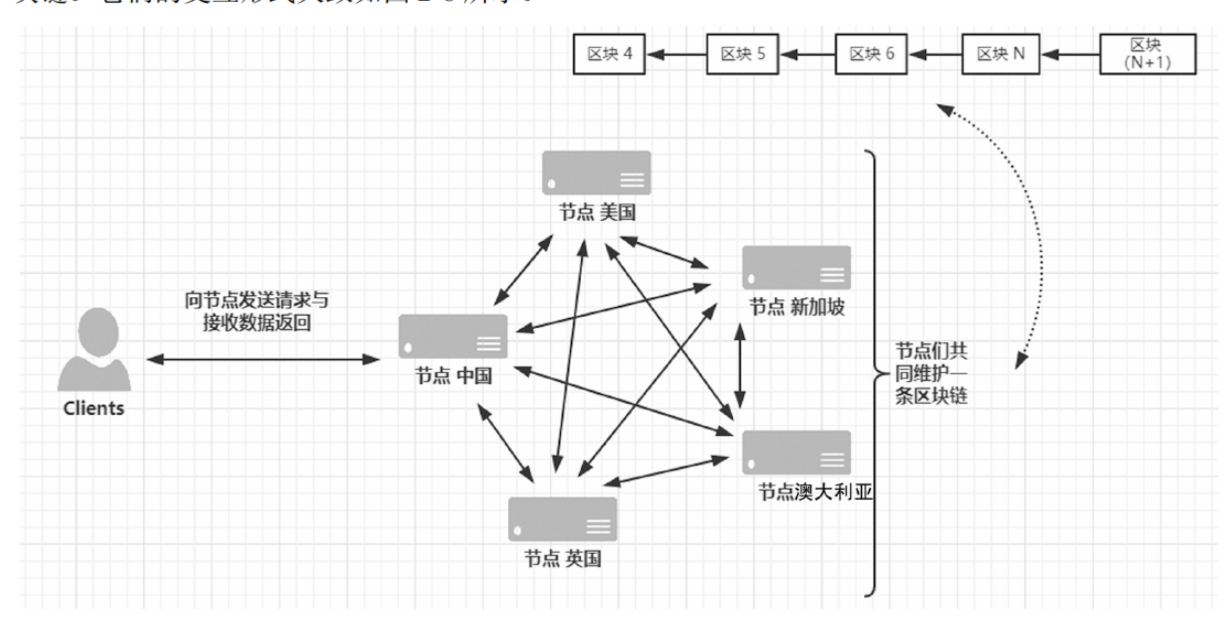



图 2-8　区块链节点去中心化集群

本书所要阐述的 DApp 就是这种区块链去中心化分布式应用。

## 2.3.2 以太坊上的 DApp

以太坊拥有图灵完备的智能合约模块，使得开发者可以先编写好智能合约代码，再到它上面部署智能合约，最终变为合约应用，也就是 DApp。

智能合约被部署到链上后是能够被访问的。为了完善 DApp 的功能，开发者可以采用计算机语言（例如 Java 或者 Go）来编写对应于当前所发布的智能合约的访问接口，用户通过访问接口访问链上的合约，得到输出的数据。

这其实也就是当前基于以太坊所开发的 DApp 的工作流程。一个 DApp 中包含多个角色，每个角色都有其各自的功能，具体说明如下：

- 智能合约应用，布置在链上，负责链上数据的处理。
- 中继服务器，布置在开发者的物理服务器上，负责接收用户的请求和访问链上的智能合约应用，再将数据结果返回给用户。

● 以太坊公链，是智能合约的集成运行环境以及实现去中心化等区块链功能的核心支撑。

图 2-9 是目前以太坊 DApp 的常见交互模型图。

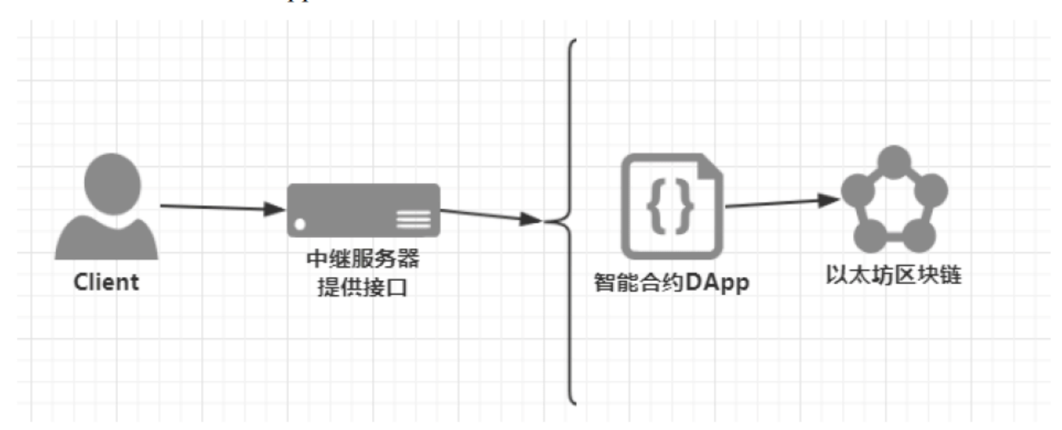



图 2-9　以太坊 DApp 的交互模型图

# 2.4　区块的组成

本节我们主要介绍以太坊公链区块（Block）的定义与组成。

## 2.4.1　区块的定义

在共识算法的伪代码中，我们已对区块做过介绍，区块其实就是一种数据结构，内含变量和属性，这些变量和属性可由开发人员自行定义。在以太坊的 1.8.11-Golang 版本代码中，给出了如下的区块定义：

```
type Block struct {
    header       *Header
    uncles       []*Header  // 这个是保存叔块头部信息的数组变量
    transactions Transactions
    // 缓存
    hash atomic.Value
    size atomic.Value

    // Td 是核心模块 core 用来存储当前区块被挖出后，区块的总难度
     td *big.Int

     ReceivedAt   time.Time    // 区块被接收的时间
    ReceivedFrom interface{} // 记录该区块是从哪个 P2P 网络传过来的
}
```

从上述区块的定义可以看到，有很多变量，现在我们不需要了解所有这些变量，除非你要深入分析以太坊的源码，但这也会花掉你大量的时间和精力，建议读者等使用以太坊做了一定的 DApp 开发之后再进行源码的全面分析。

有关上面的区块结构，建议重点了解以下部分：

- Header，区块的头部结构体。
- Transactions，当前该区块所有打包的交易记录的结构体数组。
- Hash，区块的哈希值，这个值的计算是比较复杂的，是将当前的区块头（Header 内的数据）整体地进行哈希算法运算之后所得出的哈希值，一旦区块头中某一个成员变量的数据值改变了，该哈希值就会随之改变。
- Uncles，叔块，拥有特别的含义，将会在下面的“叔块”一节中进行详细介绍。

下面我们再来看一下最主要的区块 Header 结构体的组成部分。

```
type Header struct {
    ParentHash  common.Hash
    UncleHash   common.Hash
    Coinbase    common.Address
    Root        common.Hash
    TxHash      common.Hash
    ReceiptHash common.Hash
    Bloom       Bloom
    Difficulty  *big.Int
    Number      *big.Int
    GasLimit    uint64
    GasUsed     uint64
    Time        *big.Int
    Extra       []byte
    MixDigest   common.Hash
    Nonce       BlockNonce
}
```

以上 Header 结构体中的变量基本包含了我们初步需要深入了解的全部内容，下面对需要掌握的知识逐一解析。

- ParentHash：这是当前区块的上一个区块的哈希值。请回忆一下区块的链状结构，也正是因为有这个变量的存在，后一个区块的数据里面才有了上一个区块的哈希值，从而在上下连接的层次上，体现出区块链的特点。
- Coinbase：当节点首次启动时默认配给当前节点的一个钱包地址，以太坊节点使用 PoW 共识算法挖矿产生的 ETH 代币奖励会被打入该地址。如果想使挖矿奖励进入其他账户，可以进行设置。另外，在节点控制台中直接发起交易的时候，充当 From 的也是它。
- Root、TxHash、ReceiptHash：代表的都是一棵以太坊默克尔前缀（MPT）树的根节点哈希，有关它们更深层的含义会在下面一节中进行介绍。
- Difficulty：以太坊部分代码在基于 PoW 共识情况下的挖矿难度系数，代表了区块被挖出矿的难度，这个系数会根据出块速度来进行调整。以太坊第一个区块的难度系数是 131072，后面区块的难度系数会根据前面区块的出块速度进行调整，快高慢低。
- Number：区块号，不能理解为区块的 id，因为 Number 并不是完全唯一的。例如，在几个私有节点中，每个节点会各自挖出自己节点网络中 Number 顺序递增的区块，此时的 Number 就会在不同的节点网络中出现一样的情况，而区块的 id，一般我们认为是它的区块哈希值（block hash）。另外，当前子区块的 Number，在关系方面等于在它父区块的 Number 上加 1。

- Time：区块的生成时间。请注意，这个时间不是区块真正生成的精确时间，这个时间可能是父区块的生成时间加上 N 秒，把它称为区块的大概生成时间比较准确。
- GasLimit：区块 Header 中的 GasLimit 和交易中的 GasLimit 的含义不同，请注意区分。Header 里的 GasLimit 是单个区块允许的最多交易加起来的 GasLimit 总量，即区块 GasLimit ⩾ 当前区块所有的 Transaction（交易）的 GasLimit 之和。假设有 5 笔交易，Transaction 的 GasLimit 分别是 10、20、30、40 和 50。如果区块的 GasLimit 是 100，那么前 4 笔交易就能被成功打包进入这个区块，因为矿工有权决定将哪些交易打包进区块；另一个矿工也可以选择打包最后两笔交易进入这个区块（50+40），然后将第一笔交易打包（10）。如果我们尝试将一个使用超过当前区块 GasLimit 的交易打包，这笔交易将会被网络拒绝，客户端也会收到 GasLimit 类的错误信息反馈。
- GasUsed：表示这个区块中所有的打包交易 Transaction 实际消耗的 Gas 总量，它和 GasLimit 的关系可以表示为公式：GasUsed ⩽ GasLimit。也就是说，GasLimit 虽然表示了一个总的限制值，但是实际共占了多少还是要看 GasUsed 的值。它的计算方式将会在“GasUsed 的计算”一节中进行详细讲解。
- Extra：该变量用于为当前区块的创建者保留附属信息。例如，节点 A 产生了区块 1，然后 A 向 Extra 中加上附属信息：这是节点 A 产生的区块。
- Nonce：英文解释是“临时工”，但它所表示的作用和“临时工”无半点类似。注意，Header 的 Nonce 和交易中的 Nonce 代表的含义是不一样的，Header 的 Nonce 主要用于 PoW 共识情况下的挖矿，用于记录在该区块的矿工做了多少次哈希才成功计算出胜出区块 B，例如区块 B 的 Nonce 是 200。而交易中的 Nonce 才是我们需要重点理解的，有关交易中的 Nonce 将会在下面的一节中解析。得出正确值所计算的总次数，例如 A 矿工计算了 200 次。

## 2.4.2 以太坊地址(钱包地址)

在 Header 结构体中有一个 Coinbase 变量，其本质上是一个字符长度为 42 的十六进制地址值。在以太坊中，每一个账户包括智能合约都有一个唯一标识自己的地址值，通过这个地址值可以使用以太坊的 RPC 接口查询到相关的信息。这个地址值有如下规则：

（1）0x 开头。

（2）除 0x 这两个字符外，剩下的部分必须是由字母（a~f）和数字（0~9）组成的 40 个字符。其中，字母不区分大小写。

（3）整体是一个十六进制字符串。

例如，0x24602722816b6cad0e143ce9fabf31f6012ec622 就是一个以太坊的合法地址，而 0x24602722816b6cad0e143ce9fabf31f6026ec6xy 就不是一个合法的地址，因为它最后的两位字符是 xy，不是十六进制字符。

通常，又称上面的以太坊地址为一个钱包地址，因为转账交易就是通过这个地址来转给别人的。在这一点上，类似银行卡的卡号，即银行需要银行卡的卡号才能给对方转账。

### 1. 地址的作用

地址的作用主要有下面几点：

（1）唯一标识一个账户或智能合约。
（2）作为标识，可用于查询该账户的相关信息，例如代币余额、交易记录等。
（3）进行以太坊交易时，充当交易双方的唯一标识。

参考图 2-10 所示。

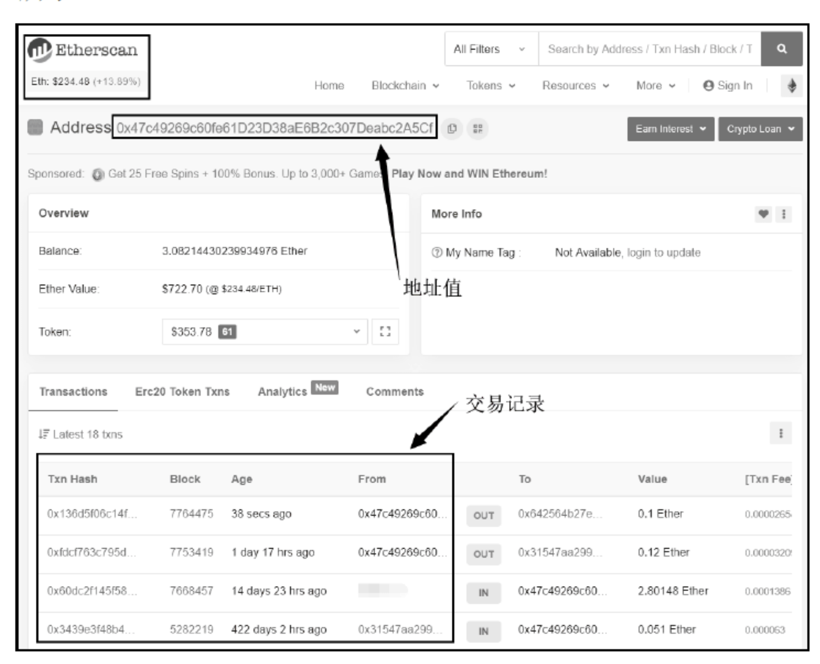



图 2-10　根据以太坊地址查询交易记录

地址分为两类：非智能合约地址与智能合约地址（又称为外部账户和合约账户）。那么如何判断一个地址是不是合约地址呢？判断方法可以使用以太坊源码提供的“eth_getCode”接口，关于该接口将会在“重要接口的含义详解”一节中讲解。

### 2. 地址的生成

地址分为合约地址和非合约地址。在以太坊的账户体系中，不同种类的地址生成方式是不同的。比如，在生成钱包地址（非合约地址）的时候，首先要根据非对称加密算法（Asymmetric Cryptographic Algorithm）中的椭圆曲线算法生成私钥和公钥，再从公钥的哈希结果中提取后 20 个字节作为非合约地址。

以太坊非合约地址（外部账户地址）的生成流程总结如下：

（1）随机产生一个私钥，32 个字节。
（2）计算得到私钥在 ECDSA-secp256k1 椭圆曲线上对应的公钥。
（3）对公钥做 SHA3 计算，得到一个哈希值，取这个哈希值的后 20 个字节来作为外部账户的地址。

智能合约地址的生成流程如下：

（1）使用“rlp”算法将（合约创建者地址+当前创建合约交易的序列号 Nonce）进行序列化。
（2）使用 Keccak256 将步骤 1 的序列化数据进行哈希运算，得出一个哈希值。
（3）取第（2）步的哈希值的前 12 字节之后的所有字节生成地址，即后 20 个字节。

非合约地址和合约地址生成方式的区别是：合约地址和椭圆曲线加密无关，因为合约地址是基于用户地址和交易序列号的，所以也不会生成雷同的地址。

大家可能还会问，为什么非合约地址要搞这么复杂还这么难读，像银行卡的卡号一样不行吗？

非合约地址之所以遵守上述的生成规则，主要原因是私钥几乎为 0 概率的重复性。私钥是通过伪随机算法（PRNG）产生的，所生成的私钥以二进制的形式表示，一共有 256 位（即 32 个字节），即 256 个 0 和 1 组成，它的可能性有 $2^{256}$ 个，此数非常庞大，比宇宙中的原子数量还要多出几十个数量级。在这种情况下，可以 100%保证账户不重复。此外，十六进制形式的地址也便于程序读写。

## 2.4.3 Nonce 的作用

上一节我们已经了解了区块结构 Header 中的 Nonce，也提到了交易中的 Nonce，它们两个是不一样的。区块 Header 中的 Nonce 主要用于 PoW 共识情况下的挖矿，交易中的 Nonce 指的是我们在调用以太坊的交易 RPC 接口进行转发操作时所要传带的参数，它代表了“交易的系列号”。

交易中的 Nonce 是相对于 from 发送者地址而言的，它代表当前发送者的账户在节点网络中总的交易序号，每个发送者地址都有一个 Nonce。from 的格式就是前面讲到的地址格式，例如 0x24602722816b6cad0e143ce9fabf31f6012ec622。

进一步举例说明：

在以太坊主网（也就是在公链）的环境下，例如账户 A，第一次进行交易，此时它的 Nonce 为 0。交易成功后，它要进行第二笔交易，此时发起交易的时候 Nonce 为 1。成功后，下一次 Nonce 为 3，一直以此类推下去。这里只考虑了每一笔都是成功的情况，事实上还有一种等待状态，此时的 Nonce 有其他的选择。另外，在不同的链和不同的节点网络中，Nonce 也不一样。

Nonce 的特点是，在顺序不断递增的交易订单中，每一次传输必须要满足比上一次成功交易的 Nonce 值要大。注意这里的一个条件，比上一次成功的交易大，其一般采取加 1 累增的方式。例如，上面的例子，在第二次发起交易的时候，Nonce 不能再为 0，否则，以太坊会返回错误，导致交易失败。可以取 3 吗？可以，但是如果取 3，必须等 Nonce 为 1 和 2 的交易被节点处理完成后才能轮到 Nonce 为 3 的交易。

因此，在每一笔成功的交易中都有一个特定的 Nonce 与之对应，这样可以有效地分辨出哪些是被重复发起的交易，以方便进行处理。

综上所述，交易中 Nonce 的作用主要有两点：

（1）作为交易接口的参数。
（2）代表每次交易的序列号，方便节点程序处理被重复发起的交易。

下面是 Nonce 的取值规则：

（1）如果 Nonce 比最近一次成功交易的 Nonce 要小，转账出错。

（2）如果 Nonce 比最近一次成功交易的 Nonce 大了不止 1，那么这次发起的交易就会长久处于队列中，此时就是等待（Pending，或称为挂起）状态！在补齐了此 Nonce 到最近成功的那个 Nonce 值之间的 Nonce 值后，此交易依旧可以被执行。

（3）还处于队列中的交易，在其他节点的缓存尚未收到并留存这次交易的广播信息的情况下，如果此时这个发起交易的节点“挂”了（就是宕机了或者脱网了），那么还没被处理的这次交易将会丢失，因为此时的交易存放于内存中尚未广播出去。

（4）处于等待（Pending）状态的交易，如果其 Nonce 相同，就会引发节点程序对交易的进一步判断，通常会选出燃料费最高的，替换掉燃料费低的（注意：前文与 Gas 相关的变量，就是指燃料，以及相关的燃料费，下一节会详细说明）。

### 2.4.4 燃料费

燃料费给我们最直观的感知就是日常生活中汽车所使用的汽油的费用，即在加油时付给加油站的油费。以太坊中的燃料费也可以这么理解（以太坊中习惯称为燃料费而不是油费）。

在以太坊中，燃料费付给的不是加油站，而是节点中的矿工。我们付给矿工燃料费的目的是让矿工帮助处理交易订单，把交易订单打包到区块中去。注意这段话中的关键词“交易订单”，不是转账，转账是交易的真子集。

以太坊中的燃料费又称为手续费，英文单词对应的是“Gas”。它是用来激励矿工把交易订单打包到区块中而付给矿工的打包费。此外，燃料费的高低会影响当前交易订单被打包的速率，高燃料费的交易订单将会优先被矿工打包进区块，以太坊使用这种价高者优先的策略保证了矿工利益的最大化。

在交易时，燃料费或手续费 Gas（单位是 wei）并不属于交易函数的参数，它的计算方式是：

```
GasUsed×GasPrice=Gas
```

我们可以采用下面的例子来进一步理解 GasUsed 和 GasPrice 的关系，假设 GasUsed 代表的是苹果的数量，那么每个苹果的单价就是 GasPrice。买了 10 个苹果，这 10 个苹果的总价格就是“数量”乘以“单价”，最终的总价就是 Gas，也就是 GasUsed×GasPrice。

公式中的 GasUsed 和 GasPrice 在以太坊中有两种解析，分别是：

（1）基于区块 Header 的解析。

（2）作为以太坊交易函数入参（Input Parameter）的解析。

第一种解析，在前文介绍 Header 区块时已介绍过。第二种解析，在以太坊交易中，当前的交易被矿工打包到区块中时，究竟要付给矿工多少 Gas 手续费，实际上和交易函数（接口）所携带的参数有关。交易订单被矿工打包进区块中时消耗的是 GasUsed，代表实际使用了多少燃料，GasPrice 指单价，两者的乘积就是 Gas 燃料费的真实值。

那么，为什么以太坊的矿工打包要消耗 Gas，直接打包不就行了吗？这里的原因如下：

（1）弥补计算机资源消耗的代价，因为以太坊公链允许每个人到上面进行交易，这些交易的背后依赖的是代码，而代码的运行自然依赖计算机资源，例如我们常见的服务器费用。

（2）防止不法分子对以太坊网络蓄意攻击或滥用，以太坊协议规定交易或合约调用的每个运

算步骤都需要收费，以增加攻击代价。

但是，请注意，以太坊上的操作并非都要扣除燃料费才能进行，常见的需要扣除燃料费的情况有：

（1）交易类型的 ETH 或 ERC20 代币转账。
（2）发布智能合约。
（3）ERC20 代币授权。

从最直观的代码的角度来看，扣除燃料费的操作有一个共同点——它们都是通过调用以太坊的“eth_sendTransaction”或“eth_sendRawTransaction”接口实现的。

此外 wei 是 Gas 的单位，它和 ETH 的对应关系如图 2-11 所示。

| 单位 | wei值 | Wei |
|---|---|---|
| wei | 1 | 1 wei |
| Kwei (babbage) | 1e3 wei | 1,000 |
| Mwei (lovelace) | 1e6 wei | 1,000,000 |
| Gwei (shannon) | 1e9 wei | 1,000,000,000 |
| microether (szabo) | 1e12 wei | 1,000,000,000,000 |
| milliether (finney) | 1e15 wei | 1,000,000,000,000,000 |
| ether | 1e18 wei | 1,000,000,000,000,000,000 |

图 2-11　燃料费单位的换算

图 2-11 中的 ether 就是一个 ETH，1eX 代表的是 10 的多少次方。

## 2.4.5　GasUsed 的计算

由燃料费的计算公式 GasUsed×GasPrice = Gas 可知，真正影响燃料费计算结果的是 GasUsed。GasUsed 的计算是比较复杂的，主要分为两部分：

（1）数据量部分，对应交易函数中的“data”入参（Input Parameter）。
（2）虚拟机（EVM）执行指令的部分。

从源码中可以看到（见图 2-12），对于每一笔交易中的数据量部分的燃料费是通过统计不同类型的字节来计算的，这部分还有一个影响计算的基础量，就是默认的燃料费数值，分下面两种情况：

```
// Per transaction not creating a contract.
// NOTE: Not payable on data of calls between transactions.
TxGas                 uint64 = 21000
// Per transaction that creates a contract.
// NOTE: Not payable on data of calls between transactions.
TxGasContractCreation uint64 = 53000
```

图 2-12　不同交易类型的燃料费起始值

（1）创建合约的交易，基础量为 53000。

（2）非创建合约的交易，基础量为 21000。

当我们在交易函数中设置的 GasLimit 比基础量还要小时，就会导致交易失败，出现的错误信息是“intrinsic gas too low”（固有的燃料太少了）；随后在这个基础量的前提下，对数据所占有的字节量计算燃料费，计算的方式也按照以太坊设置的规则，即：

（1）0 字节的收费是 4，每发现一个 0 字节，基础量累加 4。

（2）非 0 字节的收费 68，每发现一个非 0 字节，基础量累加 68。

具体设置如图 2-13 所示。

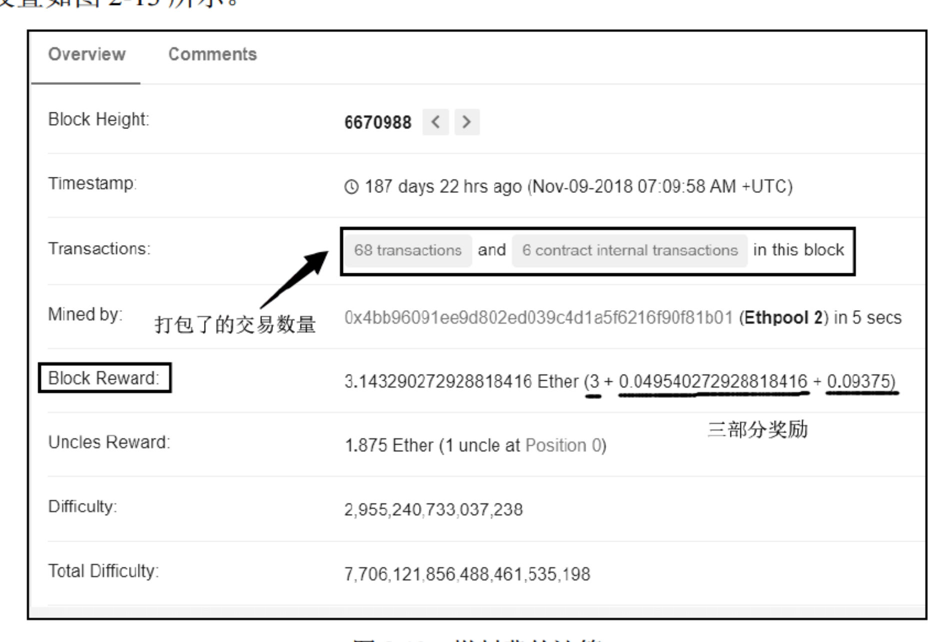



图 2-13　燃料费的计算

第一部分燃料费的计算是发生在交易被添加进订单池之前，也是在第一部分数据量所占有燃料费的数值计算结束之后。

对于第二部分的虚拟机（EVM）执行指令的计算过程发生的时机，目前有两种情况：

（1）在以太坊的矿工（Miner）模块从交易池中取出交易准备打包到区块中之前。

（2）将区块插入区块链之前需要验证区块的合法性。

如图 2-14 所示，用于计算虚拟机（EVM）燃料费的入口代码片段位于源码文件 go-ethereum\core\state_transition.go 中的 TransitionDb 函数内，该函数对不同的交易进行了对应的虚拟机操作，对于合约交易便执行“evm.Create”，对于非合约交易则执行“evm.Call”，这两种操作最终都返回了“st.gas”燃料的消耗量。

```
// Pay intrinsic gas
// 首先根据收据量和默认的 gas (53000/21000) 计算处一个基于数据量的 gas
gas, err := IntrinsicGas(st.data, contractCreation, homestead)
if err != nil {
    return ret: nil, usedGas: 0, failed: false, err
}
// st.gas - gas = 为: (Limit - 预先计算得出的 = 剩下的)
if err = st.useGas(gas); err != nil {
    return ret: nil, usedGas: 0, failed: false, err
}

var (
    evm = st.evm
    vmerr error
)
// 无论是合约的创建还是普通交易的执行，都会把上面计算了一次的 st.gas 传入到虚拟机
if contractCreation {
    ret, _, st.gas, vmerr = evm.Create(sender, st.data, st.gas, st.value)
} else {
    // Increment the nonce for the next transaction
    st.state.SetNonce(msg.From(), st.state.GetNonce(sender.Address())+1)
    ret, st.gas, vmerr = evm.Call(sender, st.to(), st.data, st.gas, st.value)
}
```

图 2-14　用于计算虚拟机燃料费的入口代码

虚拟机（EVM）执行指令部分的燃料费计算是最为复杂的。虚拟机（EVM）事务执行期间的所有操作，包括数据库读写、消息发送以及虚拟机采取的每个计算步骤都要消耗一定量的燃料，并且在不同参数和缓存影响的情况下都会对应有不同的燃料标价。图 2-15 是取自以太坊官方黄皮书中的非指令部分的燃料标价示例图。

黄皮书链接：https://ethereum.github.io/yellowpaper/paper.pdf

The fee schedule $G$ is a tuple of 31 scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

| Name | Value | Description* |
|---|---|---|
| $G_{zero}$ | 0 | Nothing paid for operations of the set $W_{zero}$. |
| $G_{base}$ | 2 | Amount of gas to pay for operations of the set $W_{base}$. |
| $G_{verylow}$ | 3 | Amount of gas to pay for operations of the set $W_{verylow}$. |
| $G_{low}$ | 5 | Amount of gas to pay for operations of the set $W_{low}$. |
| $G_{mid}$ | 8 | Amount of gas to pay for operations of the set $W_{mid}$. |
| $G_{high}$ | 10 | Amount of gas to pay for operations of the set $W_{high}$. |
| $G_{extcode}$ | 700 | Amount of gas to pay for operations of the set $W_{extcode}$. |
| $G_{balance}$ | 400 | Amount of gas to pay for a BALANCE operation. |
| $G_{sload}$ | 200 | Paid for a SLOAD operation. |
| $G_{jumpdest}$ | 1 | Paid for a JUMPDEST operation. |
| $G_{sset}$ | 20000 | Paid for an SSTORE operation when the storage value is set to non-zero from zero. |
| $G_{sreset}$ | 5000 | Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero. |
| $R_{sclear}$ | 15000 | Refund given (added into refund counter) when the storage value is set to zero from non-zero. |
| $R_{selfdestruct}$ | 24000 | Refund given (added into refund counter) for self-destructing an account. |
| $G_{selfdestruct}$ | 5000 | Amount of gas to pay for a SELFDESTRUCT operation. |
| $G_{create}$ | 32000 | Paid for a CREATE operation. |
| $G_{codedeposit}$ | 200 | Paid per byte for a CREATE operation to succeed in placing code into state. |
| $G_{call}$ | 700 | Paid for a CALL operation. |
| $G_{callvalue}$ | 9000 | Paid for a non-zero value transfer as part of the CALL operation. |
| $G_{callstipend}$ | 2300 | A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer. |
| $G_{newaccount}$ | 25000 | Paid for a CALL or SELFDESTRUCT operation which creates an account. |
| $G_{exp}$ | 10 | Partial payment for an EXP operation. |
| $G_{expbyte}$ | 50 | Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation. |
| $G_{memory}$ | 3 | Paid for every additional word when expanding memory. |
| $G_{txcreate}$ | 32000 | Paid by all contract-creating transactions after the *Homestead* transition. |
| $G_{txdatazero}$ | 4 | Paid for every zero byte of data or code for a transaction. |
| $G_{txdatanonzero}$ | 68 | Paid for every non-zero byte of data or code for a transaction. |
| $G_{transaction}$ | 21000 | Paid for every transaction. |
| $G_{log}$ | 375 | Partial payment for a LOG operation. |
| $G_{logdata}$ | 8 | Paid for each byte in a LOG operation's data. |
| $G_{logtopic}$ | 375 | Paid for each topic of a LOG operation. |
| $G_{sha3}$ | 30 | Paid for each SHA3 operation. |
| $G_{sha3word}$ | 6 | Paid for each word (rounded up) for input data to a SHA3 operation. |
| $G_{copy}$ | 3 | Partial payment for *COPY operations, multiplied by words copied, rounded up. |
| $G_{blockhash}$ | 20 | Payment for BLOCKHASH operation. |
| $G_{quaddivisor}$ | 100 | The quadratic coefficient of the input sizes of the exponentiation-over-modulo precompiled contract. |

图 2-15　指令及其对应的燃料费标价和描述

图 2-16 是指令部分的燃料费标价。

```
const (
	GasQuickStep   uint64 = 2
	GasFastestStep uint64 = 3
	GasFastStep    uint64 = 5
	GasMidStep     uint64 = 8
	GasSlowStep    uint64 = 10
	GasExtStep     uint64 = 20

	GasReturn       uint64 = 0
	GasStop         uint64 = 0
	GasContractByte uint64 = 200
)
```

图 2-16　指令部分的燃料费标价

可以看到，GasUsed 的计算在虚拟机部分是相当复杂的，要想了解完整的计算过程，建议阅读以太坊虚拟机（EVM）调用链的源码。

## 2.4.6　叔块

叔块（Uncle Block）的概念，目前只有以太坊中有。图 2-17 是在区块链分叉一节中介绍的分叉模型。

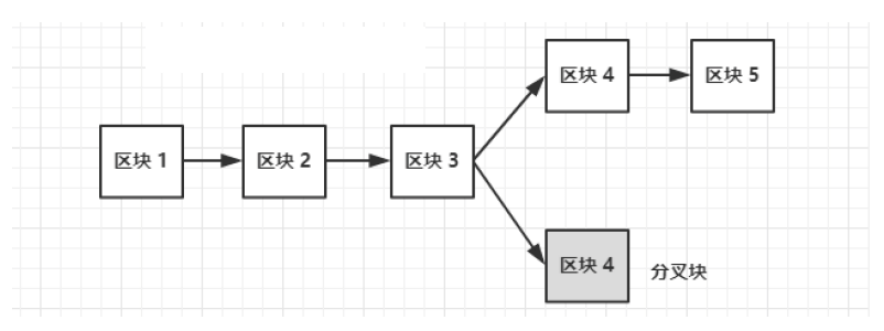



图 2-17　分叉的链

图 2-17 中导致链分叉的是分叉区块 4，根据最优链选择规则，在把造成分叉链的区块 4 抛弃后，它就成为了“孤块”，如图 2-18 所示。

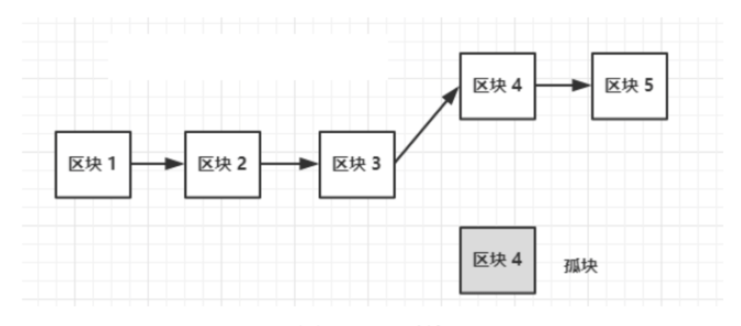



图 2-18　孤块

孤块在比特币区块链中也是存在的，因为比特币公链也有分叉的情况，但是在比特币区块链中，挖出孤块的矿工节点是得不到 BTC 奖励的，也就是没有任何奖励。而以太坊区块链不是这样的，以太坊区块链中挖出孤块的矿工也有获得 ETH 奖励的可能性，这是两者的差异。

孤块在以太坊区块链中是有机会成为叔块的，当一个孤块被另一个符合层级限制内的区块采纳为叔块的时候，挖出这个孤块的矿工就会获得以太坊 ETH 奖励，此时的孤块也就变成了叔块。

那么为什么称为“叔”而不是其他名称呢？这里实际上是类比了人类的亲戚关系。如图 2-19 所示，主链中的区块 4 和变成了叔块的区块 4 拥有相同的区块高度，就是高度 4，因为分叉的原因，导致叔块 4 最终变成了主链区块 4 的附属块，对于区块 5 来说，主链的区块 4 才是它的父区块，而由于同级关系，相对于区块 5 来说，区块 4 的附属块都是它的“叔叔”级别的块，故称为“叔块”。

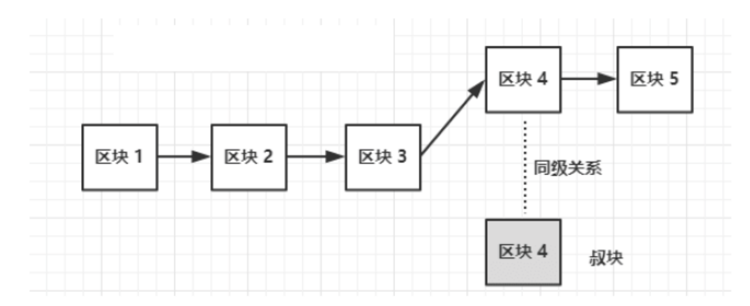



图 2-19　分叉状态的链

## 1. 打包规则

假设要将分叉区块 A 打包成为区块 B 的叔块，在代码层面就是在区块 B 的 Header 中，将区块 A 的数据绑定到区块 B 的 Uncle 变量字段中，叔块打包的规则如下。

（1）区块 A 必须是区块 B 的第 X 层祖先的同层级高度的分叉区块，2<=X<=7，参考图 2-20 所示。

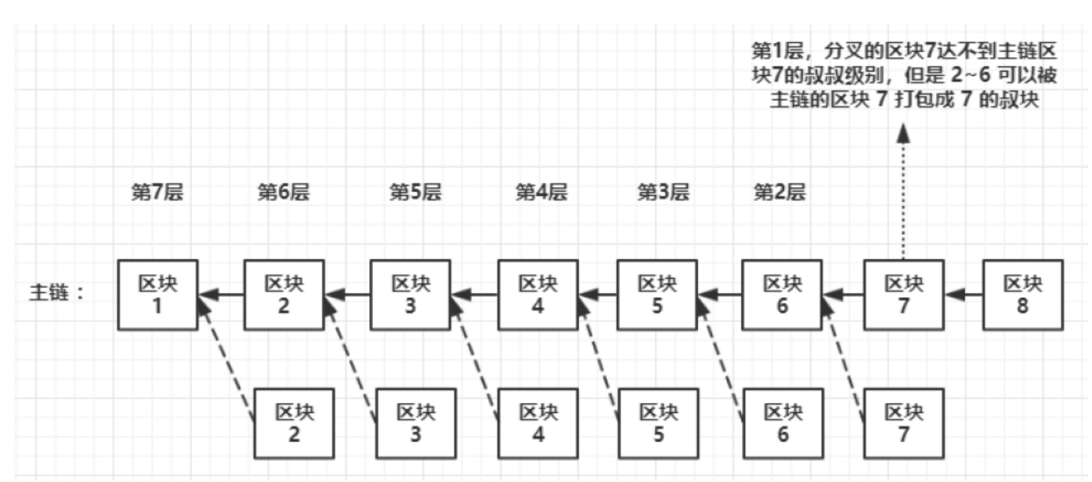



图 2-20　叔块的打包范围

（2）区块 A 必须有合法的“Block Header”（区块头部）。

（3）区块 A 必须还没成为过别的区块的叔块。

（4）区块 B 已经打包了的叔块必须还没有达到 2 个的数量，即一个区块最多只能有 2 个叔块。

（5）区块 A 一旦成为叔块，当它作为分叉区块时，打包了的交易会重新回到节点的交易池中，等待重新被打包到区块内。

### 2. 奖励规则

叔块的奖励规则是由以太坊的“Ghost 协议”（也称为幽灵协议）制定的，挖出叔块的矿工可以获得奖励。挖出叔块的矿工的奖励机制，它的奖励计算公式如下：

(uncleNumer+8-headerNumber)×blockReward/8

上述公式中的 3 个变量说明如下：

- uncleNumber：代表当前叔块的高度，也就是它的区块号。
- headerNumber：代表当前正在被打包的区块的高度。
- blockReward：代表矿工挖出区块时的基础奖励值，现在是 3ETH（君士坦丁堡版本后是 2ETH），曾经是 5ETH。
- 满足关系：headerNumber-6 < uncleNumber < headerNumber-1。

假设 headerNumber=17，那么 uncleNumber 的高度范围是 11<uncleNumber<16，奖励公式的取值范围是(2/8~7/8)*blockReward。

由于这个规则，导致了叔块的奖励也有对应的层级，当基础奖励值 blockReward 是 3ETH 的时候，它的奖励层级表如表 2-1 所示。

表 2-1 奖励层级表

| 间隔层数 | 报酬比例 | 报酬(Eth) |
|---|---|---|
| 1 | 7/8 | 2.625 |
| 2 | 6/8 | 2.25 |
| 3 | 5/8 | 1.875 |
| 4 | 4/8 | 1.5 |
| 5 | 3/8 | 1.125 |
| 6 | 2/8 | 0.75 |

叔块存在的意义有二：一是基于挖矿节点奖励回馈，二是保持生态发展平衡。以太坊为了将出块时间缩短，导致了软分叉的出现更加频繁，叔块被产生的概率就比较高，如果类似比特币的设计，对产生叔块的矿工不给予奖励，就会有很多矿工因为产生了叔块而获取不到任何奖励，积极性降低，不利于以太坊生态的发展，所以以太坊团队引入了叔块的概念。

在以太坊的挖矿过程中，叔块的奖励只是其中的一部分，完整的挖矿奖励机制将会在下一节中进行详细说明。

## 2.4.7 挖矿奖励

挖矿是应用了 PoW 共识算法的区块链所特有的操作，因为以太坊目前的共识算法主要是 PoW，所以以太坊的区块也是靠节点矿工挖矿产生的。

挖矿成功了，自然也就有收入了，这也是符合劳动获取规则的。在以太坊中，所有挖矿成功的节点都会被奖励以太坊代币 ETH，这些 ETH 将会像工资一样打入到当前节点所设置的用于收款的以太坊钱包地址中去。

打包了不同的区块就拥有不同的奖励模式，目前以太坊通过挖矿出来的区块主要有下面三种，

其中只有两种能够得到奖励：

（1）普通的成功进入主链的区块，有 ETH 奖励。
（2）被主链区块打包成叔块的分叉区块，有 ETH 奖励。
（3）孤块，没有任何奖励。

下面我们来分别认识普通区块和叔块的奖励。普通区块的 ETH 奖励由 3 个部分组成，分别是：

（1）被设置好了的固定的挖矿奖励，即“Block Reward”（区块奖励），这个值在以太坊早期的时候，节点规范设置是 5ETH，后面被设置为了 3ETH，升级到君士坦丁堡版本后变为 2ETH。
（2）挖出的区块打包了的所有交易的燃料费（Gas）总和。
（3）当前这个区块所打包了的叔块的奖励，每打包一个叔块，奖励 Block Reward×1/32。打包了 N 个叔块，则奖励是 N×Block Reward×1/32。

举个例子，假设成功地进入了主链的普通区块 A，它内部所有打包了的交易（Transaction）共有 40 个，加起来的燃料费（Gas）是 0.65 ETH，且它同时还打包了 1 个叔块，最多只能打包两个。那么此时挖出区块 A 的节点矿工，他的 ETH 收益就是（3+0.65+3/32）ETH 这么多。

下面我们通过区块链浏览器查找一下区块的详细信息，来验证一下上面的结论，如图 2-21 所示，区块信息链接是：https://etherscan.io/block/6670988 。

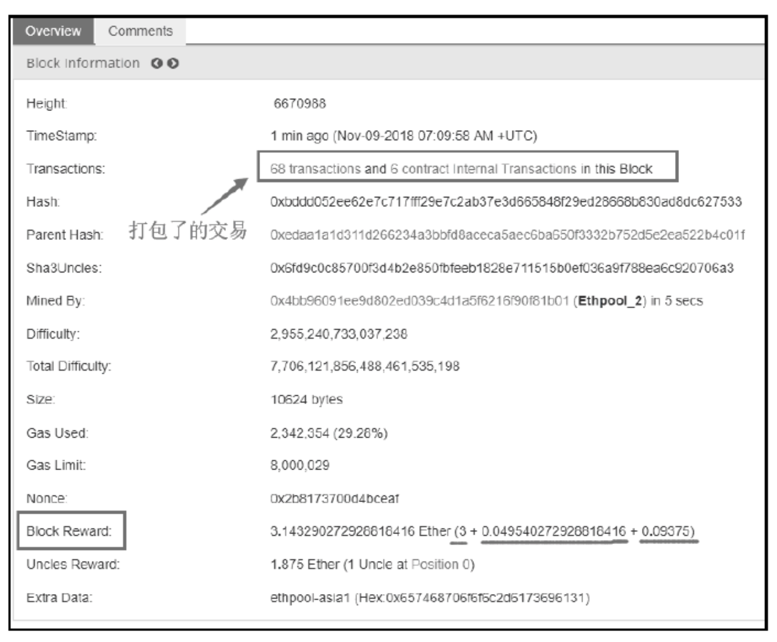



图 2-21　区块信息的展示

图 2-21 中“Block Reward”对应的就是挖出区块 6670988 的矿工，他所获得的以太坊 ETH 收入，对应前面讲解的部分，3 就是基础奖励，0.049540272928818416 是所有交易的手续费收入，这点可以单击页面中的“68 transactions”链接，进入到交易列表页面统计验证，最后的 0.09375 就是 3/32 的结果。如果该区块打包了两个叔块，那么最后的叔块收入就是 3/32×2。

图 2-21 中的“Uncles Reward”表示挖出叔块的矿工，他所获得的以太坊 ETH 收入，注意叔块奖励收入包含有下面的两种含义：

（1）挖出分叉区块的矿工的收入，因为分叉区块被其他区块打包了，它才成为了叔块。

（2）打包了分叉区块，让它成为了叔块的矿工节点的收入。

第一种奖励收入对应的奖励计算公式就是“叔块”一节中所谈到的。第二种所对应的收入就是 BlockReward×1/32×叔块个数。下面我们通过一个完整的例子来认识挖矿奖励。

假设现在有两个以太坊节点 N1 和 N2，它们所设置的挖矿收益的以太坊地址分别是 A1 和 A2，此时 N1 成功地挖出了区块 B1，高度是 4，同时 N2 也挖出了区块 B2，高度和 B1 一样，都是 4，根据最优链规则，最终区块 B2 被判断为分叉区块。紧接着，节点 N1 继续挖出了区块 B3 和区块 B4，这时属于节点 N1 的区块 B3 把区块 B2 打包成自己的叔块，使得区块 B2 不会成为孤块，而区块 B1 和区块 B4 都没有打包成叔块，如图 2-22 所示。

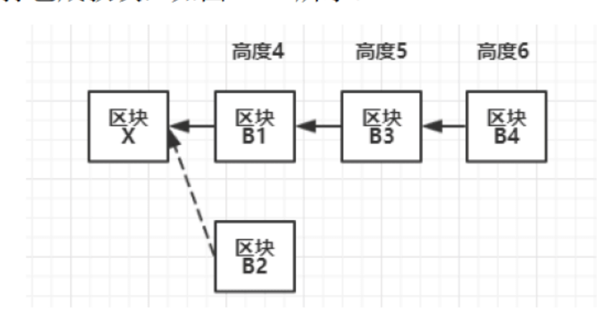



图 2-22　挖矿奖励示例

至此，节点 N1 的收益地址 A1 的总收益是：

(B1+B2+B3 的基础收益)+(B1+B2+B3 的所有打包了的交易手续费)+B3 打包了 B2 叔块的收益=BlockReward×3+T1+T2+T3+3/32

节点 N2 的收益地址 A2 的总收益是：

B2 作为叔块的奖励= (uncleNumer+8-headerNumber)×blockReward/8 = (4+8-5)×3/8 = 21/8 = 2.625，刚好对应叔块奖励表格中的第一层奖励。

# 2.5　账户模型

账户模型不仅存在于以太坊技术模块中，还存在于比特币技术模块中，它最直观的体现就是如何帮助钱包地址存储资产（代币）的数值。

在传统的服务器端的服务设计中，如果我们要为某一个用户记录他的资产余额信息，例如积分的余额，常见的做法就是直接在数据库中设置一张积分表，用来存储用户的积分。当积分有增加或使用的情况时，就对表格记录进行更新操作，表中剩下的数值就是对应的余额信息。

上面的存储做法很容易理解和接受，但是在区块链中考虑到其分布式去中心化的特点，上述的表格做法并没有被采用，取而代之的技术方案有很多，其中比较具有代表性的是比特币的 UTXO 模型和以太坊的 Account 模型，我们把这类模型称为账户模型。

下面我们主要对这两种模型所涉及的技术进行讲解。

## 2.5.1　比特币 UTXO 模型

UTXO（Unspent Transaction Output，未花费的交易输出）是一种交易数据的存储模型。目前比特币所采用的就是它。

UTXO 比较接近我们生活中钱财交易的记账模式，每一条符合 UTXO 模型的交易记录都拥有如下特点：

（1）每笔交易拥有：

- 输入部分（Input）
- 输出部分（Output）

（2）输出能够从“Unspend”状态转为“Spend”状态，这个过程称为“被使用”。“Spend”状态的输出会成为另外一条符合 UTXO 模型交易的输入部分。

（3）输出部分包含：

- 已被花费的输出，即已经当作了后面交易的输入，此时的“Spend”字段的值为 true。
- 没被花费的输出，即还没被作为后面交易的输入，此时的“Spend”字段的值为 false。

| 注　意 |
|---|
| 只有尚未花费的输出才是所谓的 UTXO——未花费的交易输出。已被花费的交易由于已经支付了，因此不是 UXTO。 |

注意区分概念：UTXO 模型不等于 UTXO 交易，后者是前者的真子集。

下面举例进一步阐述 UTXO 模型的特点。

假设一开始的时候 A 拥有 100 个 BTC，B 和 C 都拥有 0 个 BTC，如图 2-23 所示。




图 2-23　A、B、C 初始拥有的 BTC

现在 A 向 B 转账 10 个 BTC。于是，A 剩下 90 个 BTC，而 B 得到了 10 个 BTC，B 的余额是 10。此时的交易记录如图 2-24 所示，其内部包含一入一出的记录。

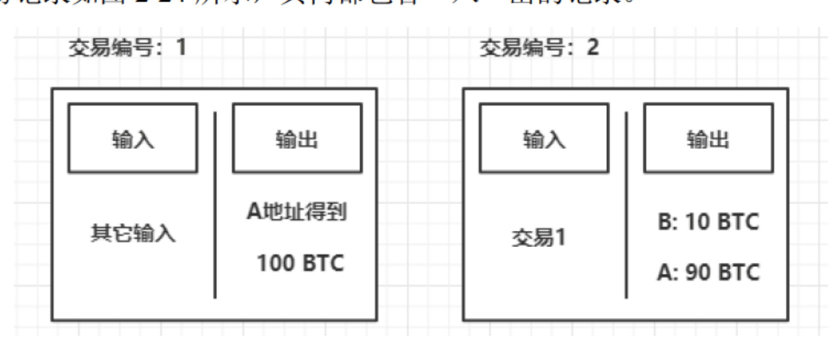



图 2-24　交易记录

A 的 100 个 BTC 不会凭空产生，它也是由其他输入赋予的，例如比特币挖矿所得。在交易 2 中，其输入部分为交易 1 的输出，此时交易 1 的输出变为“Spend”状态，交易 1 中的输出不再是 UTXO 交易，因为它已经作为了交易 2 的输入。交易 2 使用输入的 100 个 BTC，分别输出给 B 和 A。B 获得 10 个 BTC，A 进行自己的找零操作，给了 B 的 10 个 BTC 后，自己剩下 90 个。此时在交易 2 中的 B 和 A 的输出都是 UTXO，因为它们还没被“花费”。

当 A 又给 C 转账 6 个 BTC 和给 B 转账 3 个 BTC，交易记录如图 2-25 所示。




图 2-25　交易记录内部数据组成的模型

此时，交易 2 中 A 的输出将不再是 UTXO，因为它成为了交易 3 的输入；而交易 2 中 B 的依然是 UTXO，因为它还没作为其他交易的输入。

在往后的交易中便一直按照上面的记录形式来记录交易的输入和输出。

在上面的比特币例子中，UTXO 模型存在几种情况，但不变的是其记录始终由输入和输出组成，在此之外还多了一个手续费的概念。一般地，输入和输出拥有下面的几种组合情况：

- 输入的条数比输出的多，输出的条数不只一条。
- 输出的条数比输入的多，输入的条数不只一条。
- 设 sum 是累计、输入数值为 inputs、输出数值为 ouputs、手续费是 fee，那么比特币中的一笔交易恒满足：sum(inputs) - sum(ouputs) - fee = 0。

比特币最初的代币产生是从挖矿中获取的，后续的代币因为不断地被交易而被分配到各个地址中去，根据 UTXO 的模型，可以知道：

（1）每一笔交易的输出最终都能追寻一个一开始的输入。

（2）交易的最初输入都来源于挖矿的收益地址，这个地址我们一般称为“CoinBase”。

那么如何统计一个地址的 BTC 余额呢？其实就是统计其 UTXO 集合。在图 2-25 中，B 的 BTC 余额相关的 UTXO 分别在交易 2 和交易 3 中，为 10+3=13 个 BTC。

在区块链中，不同的交易会被打包到不同的区块中去。这意味着，当我们需要计算某个地址中的余额时，需要遍历整个网络中所有与该地址相关的区块内的 UTXO，汇总后便是它的余额。

UTXO 模型明显的优点：

- UTXO 模型是无状态的，只要交易的签名合法，交易额正确，那么当交易被区块打包并广播确认后就会被直接进行存储。这会更容易应对并发转账的情况，因为没有类似序列号的东西，当一个地址拥有很多 UTXO 的时候，可以同时发起多笔交易。
- 除 CoinBase 交易外，交易的 Input 始终链接在某个 UTXO 后面，交易的先后顺序和依赖关系

容易被验证。

UTXO 模型明显的缺点：

- 无法实现比较复杂的逻辑，可编程性差。例如，以太坊的智能合约功能就无法通过 UTXO 模型进行拓展实现。
- 性能问题，例如计算某个地址中的余额时，需要遍历整个网络中的全部相关区块，找到该地址的 UTXO，当该地址相关的交易遍布区块较多时，时间复杂度将会剧增，获取余额的操作会出现比较慢的情况。

## 2.5.2　Trie 树

### 1.Trie 树的定义

Trie 树又称字典树，是树形数据结构中的一种，同范畴的还有完全二叉树、红黑树等，如图 2-26 所示。

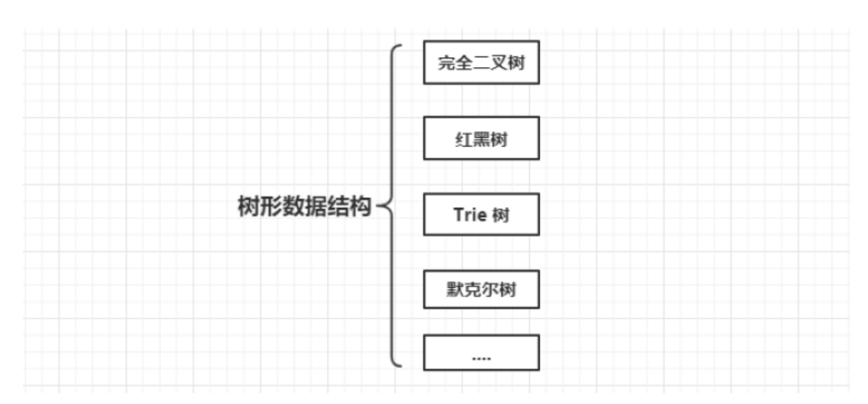



图 2-26　树形数据结构的分类

Trie 树的检索体现在它使用数据某种公共前缀作为组成树的特点，下面举例说明。假设有英文组合词 taa、tan、tc、in、inn、int，这些词就是我们的数据。分析它们的前缀特点：首先 taa、tan、tc 这 3 个单词拥有公共的开头字母 t，这就是它们的公共前缀，归为一类；然后 in、inn、int 的公共开头字母是 i，根据这个特点，我们得到如图 2-27 所示的树形图。

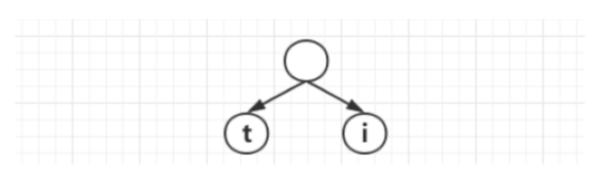



图 2-27　树形图一

接着继续分析，在 taa、tan、tc 中标出第一个字母 t 后，剩下的分别是 aa、an、c。可以看到，aa 和 an 拥有公共的前缀字母 a，因此模仿上面的前缀规则可以继续完成树形图，如图 2-28 所示。

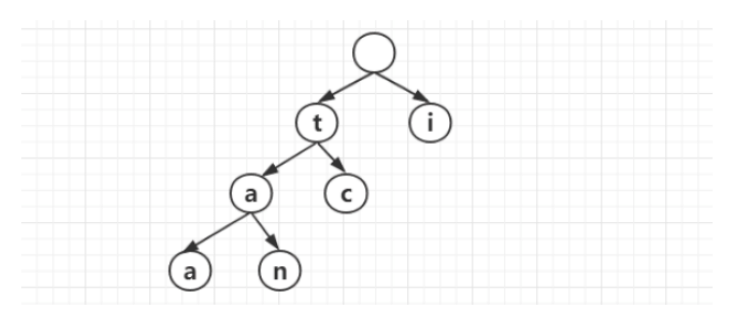



图 2-28 树形图二

至此，从树顶节点开始自上而下看，所走过的节点值连起来就是 taa、tan、tc。继续完善前缀 i 字母的子树，最终整个 Trie 的前缀树如图 2-29 所示。

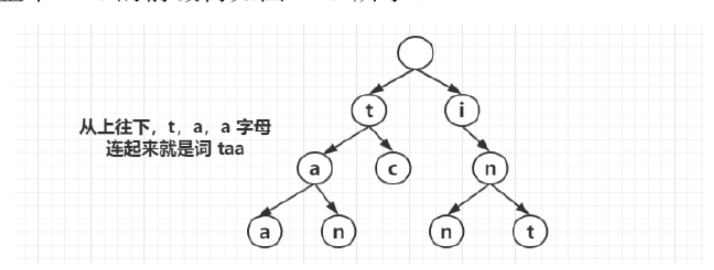



图 2-29 Trie 的前缀树

如果例子中在起始的时候多出一个和其他节点没有公共前缀的单词，例如 egg，那么树图从顶部开始将分成 3 个分支，分别是 t、i、e，而不是两个。

最终，各个单词被包含在 Trie 树中。一棵 Trie 树满足下面的特点：

- 不一定是二叉树。
- 根节点不包含字符，除根节点以外每个节点只包含一个字符，注意是字符不是字符串。
- 从根节点到某一个节点，自上而下，路径上经过的字符连接起来就为目的节点对应的字符串。
- 每个节点的所有子节点包含的字符串不相同。

### 2. Trie 树的应用

为什么要在软件应用中采用 Trie 树这种数据结构呢？这是因为 Trie 树在针对字符串搜索方面有很好的性能。

接着 Trie 树的例子，如果我们要查找 tan 这个单词，可以按照下面的步骤来执行。

（1）首先自上而下，先查找字母 t，如果找到了 t，那么不是 t 的分支就不需要考虑了。
（2）接着查找字母 a，以此类推。
（3）最终找剩下的字母 n。

在上述查找过程中，最大限度地减少了无谓字符的比较，但由于 Trie 树的非根节点存储的是每一个字符，导致 Trie 树会消耗大量的内存，这也是 Trie 树的一个缺点。此外，Trie 树中由于字符串之间没有公共的字母前缀，因此树的层级也会比较高，比如说 taa 和 tcn，它们只有 t 字母是公共的，那么如果是 t->aa 和 t->cn 就只有两层的高度，而在 Tire 树中，却被表示为了 t->a->a 和 t->c->n，拥有 3 层的高度。

## 2.5.3　Patricia Trie 树

Patricia Trie 树也是一种 Trie 树。不同点在于，它是 Trie 树的升级版，在 Trie 树的基础上做了优化：非根节点可以存储字符串，而不再仅仅是字符，节省了空间的花销。

仍然以上一节的 Trie 树为例，我们画出 Patricia Trie 树的树形图，单词是 taa、tan、tc、in、inn、int，如图 2-30 所示。

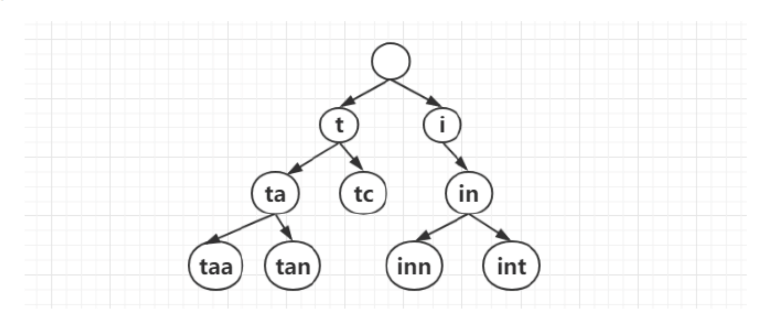



图 2-30　Patricia Trie 树的树形图

这里我们给出 abcd 和 aoip 两个字符串的 Patricia Trie 树和 Trie 树的树形图，如图 2-31 所示。可以明显地看到，那些很长但又没有公共节点的字符串在 Patricia Trie 树中占用的空间更少。

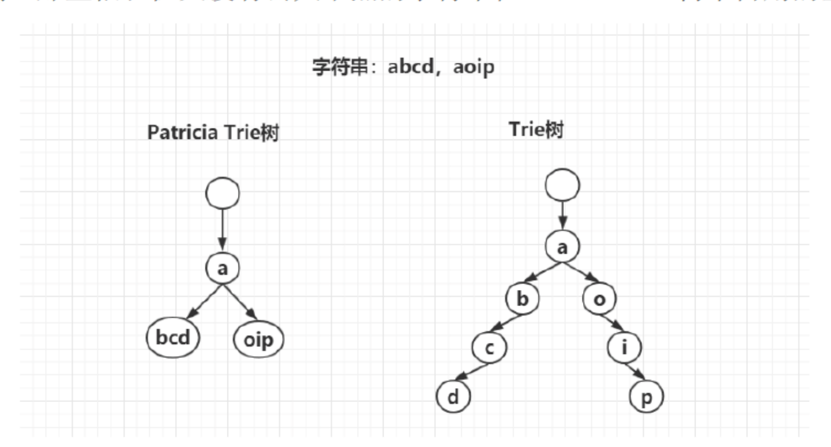



图 2-31　Patricia Trie 树和 Trie 树的树形图

## 2.5.4　默克尔树（Merkle Tree）

我们知道，以太坊区块 Header 内部的 Root、TxHash、ReceiptHash 代表的都是以太坊默克尔前缀（MPT）树的根节点的哈希值（Hash）。关于 MPT 树将在下一节中介绍，本节我们先来认识默克尔树以及这三个哈希相关的概念。

默克尔树又被称为哈希树（Hash Tree），它满足树的数据结构特点，拥有下面的特点，也就是说，默克尔树必须满足下面的条件。

- 树的数据结构，常见的是二叉树，但也可以是多叉树，它具有树结构的全部特点。
- 基础数据不是固定的，节点所存储的数据值是具体的数据值经过哈希运算后所得到的哈希值。
- 哈希的计算是从下往上逐层进行的，就是说每个中间节点根据相邻的两个叶子节点组合计算

得出，根节点的哈希值根据其左右孩子节点组合计算得出。

- 最底层的节点包含基础的数据。

图 2-32 是一棵二叉树形态的默克尔树。

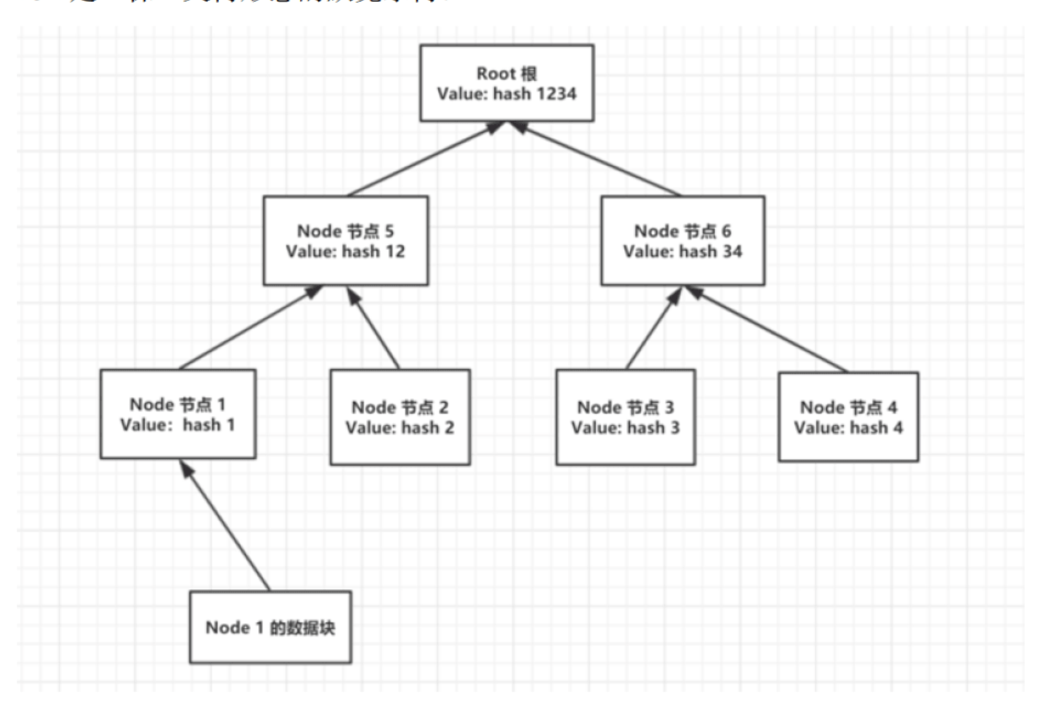



图 2-32 二叉树形态的默克尔树

（1）自下而上地看，最底层节点 Node 节点 1 的数值 Value 是 hash 1，hash 1 是由 Node1 对应的数据块经过一定的哈希算法生成的，其他的最底层节点也有对应的数据块。此处对应默克尔树的第 4 个特点。

（2）Node 节点 5 是 Node 节点 1 和 Node 节点 2 的父亲节点，那么 Node 节点 5 的哈希值由 Node 节点 1 和 Node 节点 2 的哈希值得出。具体父节点的值如何计算，并没有统一的方法，可以定义某一种算法，只要满足父节点的值为其左右叶子节点的值经过一定计算得出即可。图 2-32 采用了字符串拼接的计算方式：Value(5) = Value(1) + Value(2) = 12。此条对应默克尔树的第 3 个特点。

（3）由于生成哈希值的原始数据几乎都是字节流，因此底层数据块的内容不会被限制，类似于区块头，拥有多种数据类型，也可以是单独的一个字符串。此条满足默克尔树的第 2 个特点。

（4）我们从图 2-32 中可以很直观地看出，该默克尔树就是数据结构中的二叉树模型。

### 1.默克尔树的节点插入

图 2-32 是一种完全二叉树的形式。在此类二叉树中，当一个新的数据块产生的哈希值形成的新的节点要插入树中时，如果所要被插入的默克尔树底层的节点已经是满叶子的情况，它会按照如图 2-33 所示的形式插入。

在这种情况下，新插入的叶子节点会自动在不同的层数生成与最底层新插入的节点所拥有相同数值的节点，图 2-33 新插入节点为 A，据此依次生成 B、C、D，最后的 D 节点是新的根节点（Root）。

至此，我们知道，区块 Header 内部的 Root、TxHash、ReceiptHash 这 3 个值的含义其实都是默克尔树的根，它们所在的树依次对应于：

（1）区块体内的账户（Account）对象数组。在打包交易中该对象数组会时刻被更新。

（2）被打包进当前区块的交易（Transaction）列表数组。该列表数组在所有交易打包完之后生成。

（3）区块内的所有交易（Transaction）完成之后生成的一个 Receipt 数组。

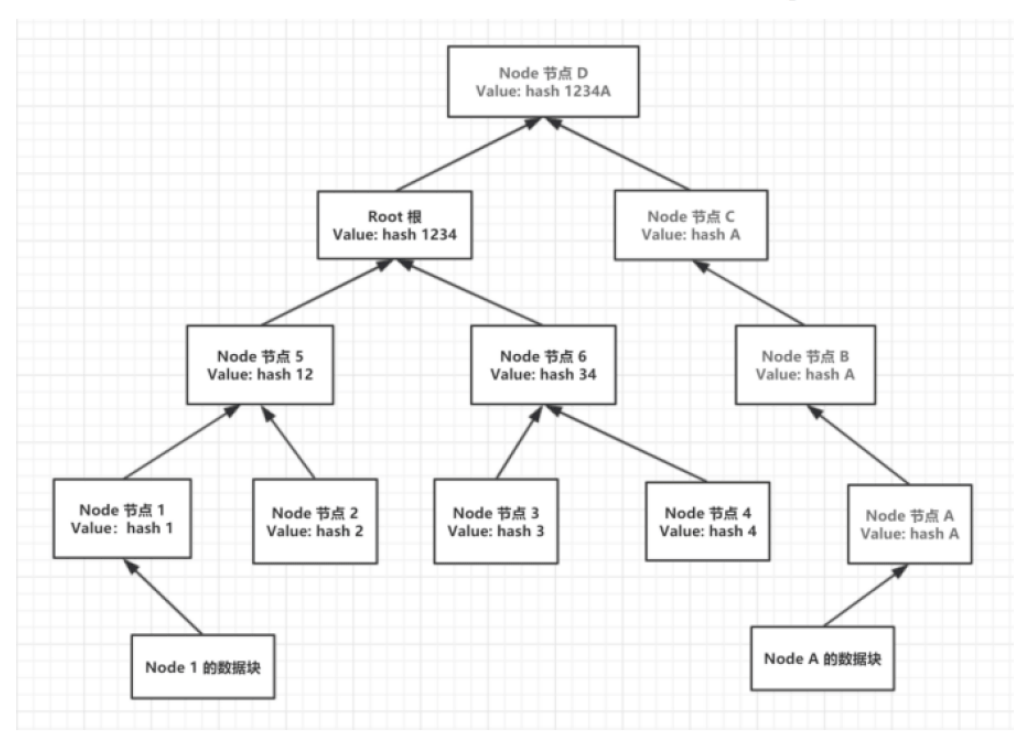



图 2-33　满叶子时在默克尔树中插入节点后的变化

## 2.默克尔树数据验证

默克尔树的作用体现得最多的地方就是它可被用于数据的验证。在以太坊中，默克尔树可以用来验证区块内的交易（Transaction），因为以太坊的交易是被矿工打包进到区块中的，所以一个区块内部包含有很多笔交易信息。

根据默克尔树父节点的哈希值与其叶节点值的关系，如果当前默克尔树的底层数据块是交易数据，那么往上的节点中，其所包含的哈希值都是由交易数据生成的。

根据节点中哈希值的关联关系，可以对某笔交易数据进行验证，如图 2-34 所示。

假设我们知道了交易数据 1、Node 节点 1 和 Node 节点 6，现在要验证交易数据 2 是否在当前的默克尔树中。首先由交易数据 1 和交易数据 2 生成 Z 节点的哈希值，然后由 Node 节点 1 和 Z 节点生成 Y 节点的哈希值，最后由 Y 节点和 Node 节点 6 生成根节点 X 的哈希值。在得到了根节点 X 的哈希值之后，再将它和区块头部中的 TxHash 值进行比较，判断它们是否相等，如果相等，证明交易数据 2 存在于当前区块的交易列表中，反之则不是。

默克尔树交易数据的验证应用还存在于点对点的视频流中。例如，将一部完整影片的数据流拆分成多个数据块，并由这些数据块组成默克尔树。当用户下载影片时，就能根据节点值来对应下载自己所缺少的那一部分，在数据被损坏的时候也能进行下载修复，而不需要重新下载整部影片。

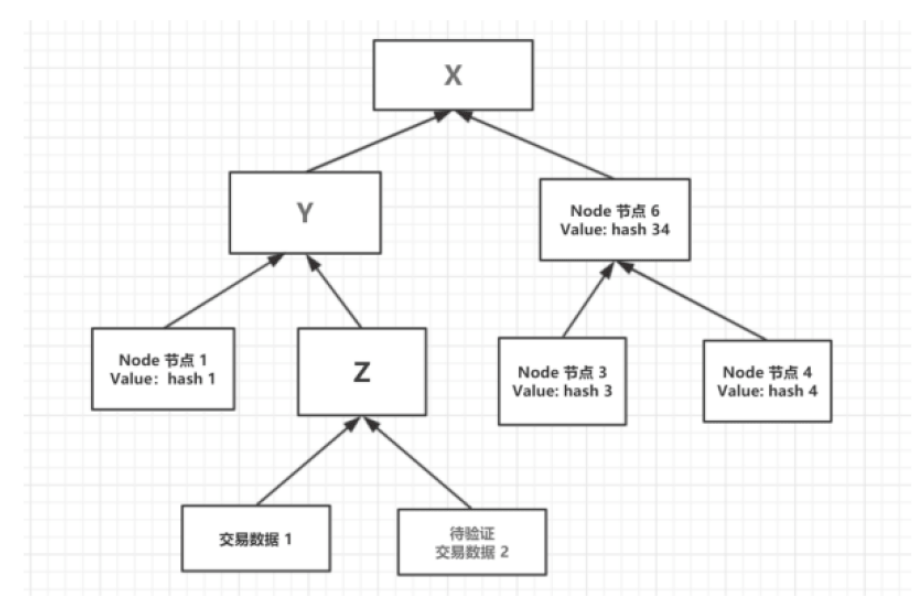



图 2-34 默克尔树交易数据的验证

## 2.5.5 以太坊 MPT 树

MPT 树的全称是“Merkle Patricia Trie”，即默克尔前缀树。根据我们对默克尔树和前缀树的介绍，MPT 树可以认为是默克尔树和前缀树的结合。事实也是如此，MPT 树是以太坊结合了默克尔树和前缀树的特点而发明的一种非常重要的数据结构，因此它具备了默克尔树和前缀树的特点。

MPT 树中的节点有以下 4 种类型：

- 扩展节点（Extension Node），只能有一个子节点。
- 分支节点（Branch Node），可以有多个节点。
- 叶子节点（Leaf Node），没有子节点。
- 空节点，空字符串。

1.节点的定义及说明

（1）扩展节点

在代码中，扩展节点含有 Key、Value 和 nodeFlag 字段变量，定义如下：

```
type shortNode struct {
        Key   []byte
        Val   node
        flags nodeFlag
}
```

（2）叶子节点

和扩展节点一样，也包含是 Key、Value 和 nodeFlag 字段变量，定义如下：

```
type shortNode struct {
        Key   []byte
        Val   node
```

```
        flags nodeFlag
}
```

（3）分支节点

分支节点的内容是一个长度为 17 的数组，其中，前 16 位每个下标的值是十六进制的 0~f、十进制的 0~15，它们每位可能指向一个孩子分支，且允许不做任何指向，在最后的第 17 位是它自己的 Value。因此，一个分支节点的孩子至多有 16 个。代码中的定义是：

```
type fullNode struct {
        Children [17]node
        flags    nodeFlag
}
```

这 3 类节点的结构可参考图 3-35。

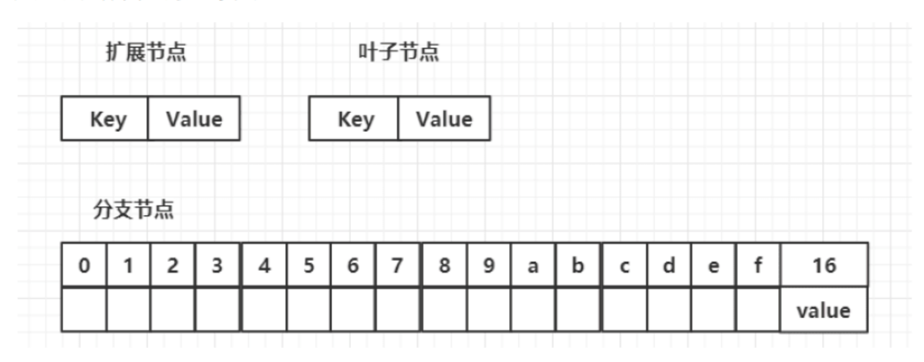



图 2-35　MPT 树中的 3 类主要叶子节点的结构

2. 节点字段变量的解释

（1）Key 只在扩展节点和叶子节点中存在。请注意，分支节点没有 Key。Key 就是[Key,Value]“键-值对”数据结构中的“键”。在以太坊中，不同的存储阶段，Key 的值是不同的，有如下一些情况：

- Raw 编码。这种编码方式的 Key 是 MPT 对外提供接口的默认编码方式。例如，一个 Key 为“cat”，则其 Raw 编码就是['c', 'a', 't']，转换成 ASCII 编码的表示方式就是[63, 61, 74]。
- Hex 十六进制编码。这种是 MPT 对内存中树节点的 Key 进行的编码方式，当数据项被插入到 MPT 树中时，Raw 编码被转换成 Hex 编码。其诞生的原因是为了减少分支节点孩子的个数，由分支节点的定义和对应的示范图可以看出，分支节点最多有 16 个孩子节点。导致最多只有 16 个孩子节点的原因，就是使用了这种编码方式，不然的话会由于原来 Key 的 8 位范围取值是[0-(2^7-1)]，即[0-127]，意味着有 128 个位，从而对应到 128 个孩子节点这么多。为了减少可对应的孩子节点数，以太坊将原 Key 的高低共 8 位分拆成两个字节，以 4 位进行存储，而 4 位在十六进制中，最大能表示的是 f，即其范围为[0-f]，从而减小了每个分支节点的容量，但是在一定程度上增加了树的高度。

  从 Raw 编码向 Hex 编码的转换规则是:

  - 将 Raw 编码的每个字符根据高 4 位、低 4 位拆成两个字节。
  - 若该 Key 对应节点存储的是真实的数据项内容（该节点是叶子节点），则在末位添加一个 ASCII 值为 16 的字符作为终止标志符。

- ➢ 若该 Key 对应的节点存储的是另外一个节点的哈希索引（该节点是扩展节点），则不加任何字符。
- ➢ 例如，某叶子节点的 Key 为“cat”，其 Raw 编码就是['c', 'a', 't']，转换成 ASCII 表示方式就是[63, 61, 74]，其 Hex 编码为：

  [0011,1111,0011,1101,0100,1010,0x10] ==> [3, f, 3, d, 4, a, 0x10] ==> [3,15,3,13,4,10,16]

- HP 编码，全称为“Hex-Prefix 编码”，即十六进制前缀编码，是 MPT 中的树节点被持久化存储到数据库层面时 Key 被编码的形式。当树节点被加载到内存中时，HP 编码会被转换成 Hex 编码，对应从 Hex 编码到 HP 编码，刚好是一个对称的过程。

  HP 编码的规则如下：

  ① 若输入 Key 结尾为 0x10，则去掉这个终止符。

  ② Key 之前补一个四元组，从右往左，这个四元组第 0 位作为区分奇偶信息，若 Key 长度为奇数则该位为 1，若长度为偶数则该位为 0。第 1 位区分节点类型，叶子节点类型是 1，其他是 0。

  ③ 如果输入 Key 的长度是偶数，就再添加一个四元组 0x0000 在第②点的四元组之后。

  ④ 将原来的 Key 内容压缩，共 8 位，以高 4 位低 4 位进行合并输出。例如，某叶子节点的 Key 为“cat”，它的 Hex 编码是[3,15,3,13,4,10,16]，根据第①点，因为 16 对应的十六进制表示就是 0x10，所以去掉它，此时变为[3,15,3,13,4,10]，共 6 个数值，所以长度是偶数。根据第②点，Key 之前补全四元组 0x0，此时变为[0x0000,3,15,3,13,4,10]，因为是节点类型，所以从右往左，第 0 位为 0，第一位是 1，变为 [0x0010,3,15,3,13,4,10]。根据第③点，Key 的长度是偶数，则再添加一个四元组 0x0 在之前的四元组之后，变为[0x00100000,3,15,3,13,4,10]。根据最后一点，压缩合并，变为[32, 63, 61, 74]，32 就是二进制 00100000 的十进制数：2^5=32。此时再转为 Hex 编码就是[2,0,3,15,3,13,4,10]。

因为在 HP 编码情况下的 Key 加入了 Prefix 前缀，所以在细分 Key 内容的时候应该多出一个前缀码，如图 2-36 所示。前缀的好处之一是能够标识这个节点的类型。

| Key-prefix | Key-end | Value |
| --- | --- | --- |
| 1 | key 内容 | Value |

图 2-36　HP 编码时的扩展节点/叶子节点

在前面叶子节点的 Key 为“cat”的例子中，通过 HP 编码计算出的[2,0,3,15,3,13,4,10]，其 Key-prefix 就是 2，Key-end 是 0,3,15,3,13,4,10。

（2）Value 是用来存储节点数值的，不同的节点类型，Value 对应的值也不同，主要有下面几种情况：

- 叶子节点。Value 存储的是一个数据项的内容，例如[name, LinGuanHong]，Key 是 name，Value 是 LinGuanHong，在代码中对应于 ValueNode。
- 扩展节点。Value 存储的是其孩子节点在数据库中存储的哈希值，可以通过该哈希链接到其他节点。在代码中，对应 hashNode 类型。
- 分支节点。Value 存储的是在当前分支节点结束时节点的数据值，在代码中对应于 ValueNode。

比如：Key 有 abc、abd、ab，根据前缀树的特点开始构建树，如图 2-37 所示。因为 3 个 Key 拥有公共的前缀 ab，其中 abc 和 abd 还多出一个字符，可以对应在分支节点中，ab 没有多出的字符，它刚好在分支节点中结束，此时分支节点的 Value 存储的就是 Key=ab 节点的值。当没有节点在分支节点中结束时，那么分支节点的 Value 没有数据存储。

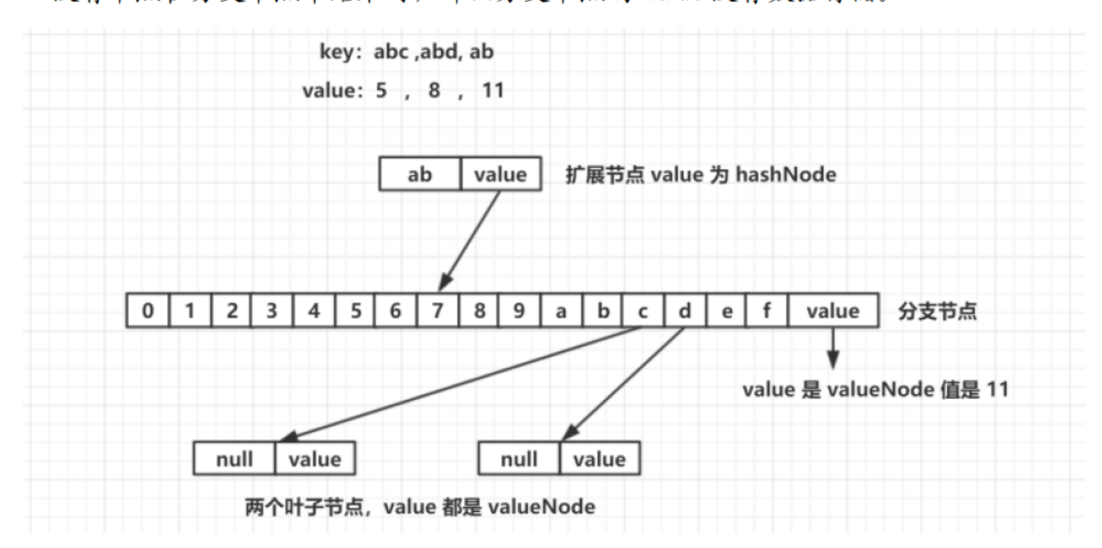



图 2-37　分支节点 Value 有值的情况

（3）nodeFlag 是分支节点、扩展节点和叶子节点在代码结构体中附带的字段，主要用于记录一些辅助数据，其代码中的定义如下：

```
    type nodeFlag struct {
        hash  hashNode // cached hash of the node (may be nil)
        gen   uint16   // cache generation counter
        dirty bool     // whether the node has changes that must be written to the
database
    }
```

说明：

- 节点哈希 hash。若该字段不为空，则当需要进行哈希计算时，可以跳过计算过程而直接使用上次计算的结果（当节点变脏时，该字段被置空）。
- 脏标志 dirty。当一个节点被修改时，该标志位被置为 1。
- 诞生标志 gen。当该节点第一次被载入内存中（或被修改时），会被赋予一个计数值作为诞生标志，该标志会被作为驱除节点的依据——清除内存中“太老”的未被修改的节点，防止占用的内存空间过多。

## 2.5.6　MPT 树节点存储到数据库

MPT 树节点存储到数据库，又称节点的持久化，这个过程需要计算出各个节点对应的 RLP 编码数据及节点的哈希值，其最终存储在“键-值对”<k,v>数据库中的格式是：[节点哈希值,节点的 RLP 编码]。要注意区分，这里持久化的哈希值不是 Key 的哈希值，而是节点 RLP 编码的哈希值。此外，持久化的计算过程是一个递归过程，意味着这个计算是从 MPT 树的底部开始从下往上进行

的。持久化的步骤是：

（1）使用“RLP”将节点的数据进行序列化编码。

- 对叶子 / 扩展节点来说，该节点的 RLP 编码就是对其 Key 和 Value 数据一起进行编码。即 rlp(Key + Value)。
- 对于分支节点，该节点的 RLP 编码是对其孩子列表对应的哈希值一起进行 RLP 编码，如果此时的分支节点的 Value 对应的是 valueNode，即有数值，那么 RLP 编码也要加入 Value，即 rlp(childNode's hash + Value)。

（2）在每个节点计算出各自的 RLP 编码后，再根据 RLP 编码计算出节点的哈希值。使用的是 SHA256 算法计算，即 hash = sha256(rlp 数据)。

（3）对应<k,v>数据库中 k=hash、v=rlp 编码，进行节点的持久化存储。

持久化对应源码中的操作代码（代码文件位置是 trie/hasher.go）如下所示：

```
func (h *hasher) store(n node, db *Database, force bool) (node, error) {
    ...
    // rlp.Encode 将节点 node 数据进行 rlp 编码，存储于 tmp 内，其中 Key 和 Value 都在内部
    if err := rlp.Encode(&h.tmp, n); err != nil {
        panic("encode error: " + err.Error())
    }
    ...
    if hash == nil {
        hash = h.makeHashNode(h.tmp) // 使用 sha256 对 rlp 数据进行哈希计算
    }
    if db != nil {
        db.lock.Lock()
        hash := common.BytesToHash(hash)
        db.insert(hash, h.tmp)  // 存储
        ...
    }
    ...
}
```

## 2.5.7 组建一棵 MPT 树

根据对 MPT 树的介绍，本节我们从插入第一个节点开始组建一棵 MPT 树，来对 MPT 树做一个整体的认识。因为节点中的 Key 在不同阶段对应的编码形式并不相同，为了体现出“Hex-Prefix 编码”，我们下面在构建的时候将 HP 编码加入到里面去。注意，在实际情况中，HP 编码只有在节点持久化时才会用到，并出现“Key-Prefix”，而在内存层面的 MPT 树，节点的 Key 是“Hex 编码”格式，此时还没有“Key-Prefix”。

用于构建 MPT 树的节点如图 2-38 所示。

首先设根节点为 Root。在构建的过程中，一般 Root 还没有生成，只有在整棵树都构建完成后才会从底部往上开始计算哈希值，最终算出根 Root 的哈希值。

插入第一个节点<a711355,45> 的时候，树如图 2-39 所示。

| 顺序 | 已经转为了16进制的key | 数值 |
|---|---|---|
| 0 | a711355 | 45.0 |
| 1 | a77d337 | 1.00 |
| 2 | a7f9365 | 1.1 |
| 3 | a77d397 | 0.12 |

图 2-38　用于构建 MPT 树的节点数据

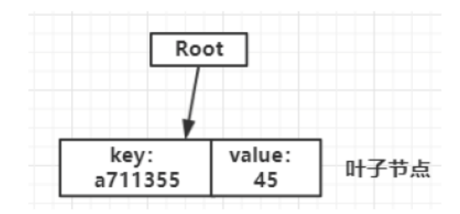



图 2-39　插入节点 <a711355,45> 的树

接着插入第二个节点<a77d337,1>。因为 a77d337 和 a711355 拥有公共的前缀 a7，所以 a7 变为一个扩展节点，其 value 存储的是分支节点的哈希值，剩下的是两个叶子节点，树如图 2-40 所示。

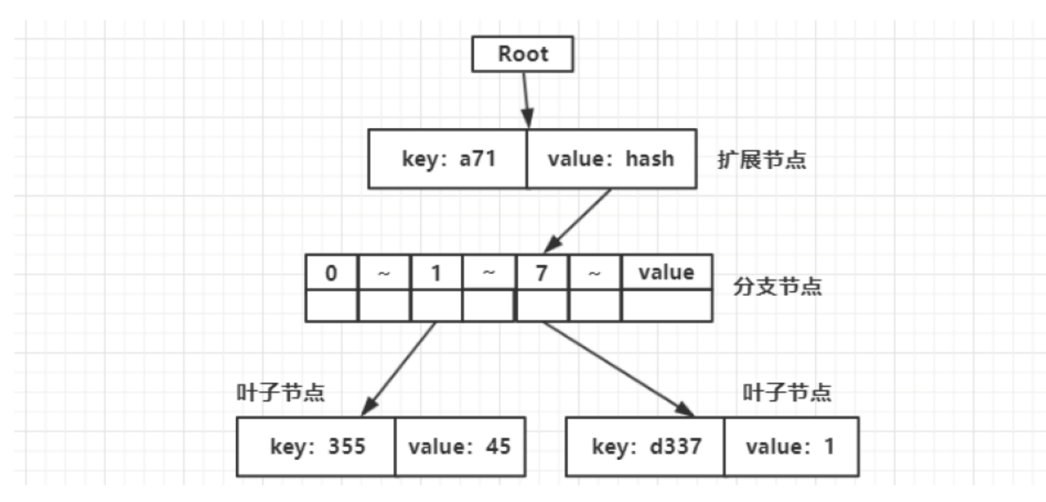



图 2-40　插入节点 <a77d337,1>之后的树

接着插入第三个节点<a7f9365,1.1>。因为这个节点和前两个节点都拥有前缀 a7，所以它将会是分支节点中的一员，插入第三个节点之后的树如图 2-41 所示。

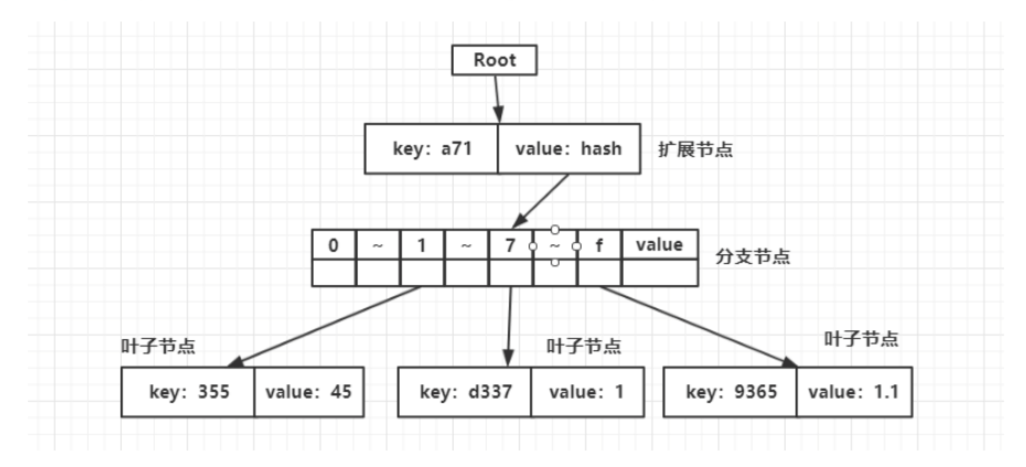



图 2-41　插入节点<a7f9365,1.1>之后的树

最后插入节点<a77d397,0.12>。因为 a77d397 和第二个节点的 key（a77d337）在分支节点之后

还存在 d3 的公共前缀，因此在它们之间要添加新的以 key 为 d3 的扩展节点，然后剩下的 37 与 97 还要添加一个分支节点。为什么是分支节点呢？因为扩展节点只能有一个孩子节点，而且我们现在还剩下 37 与 97，所以为了容纳两个节点只能使用分支节点。最终的树如图 2-42 所示。

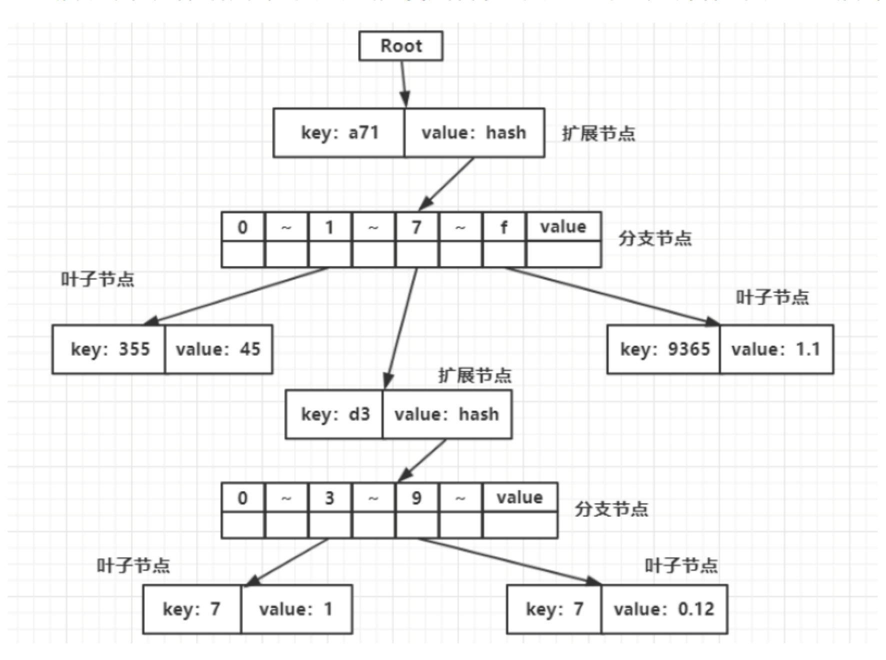



图 2-42　插入节点 <a77d397,0.12>之后的树

图 2-42 是最终构建好的 MPT 树。现在我们继续根据“Hex-Prefix 编码”计算出扩展节点和叶子节点的“Key-Prefix”值。根据 HP 编码规则，最终得到的树如图 2-43 所示。

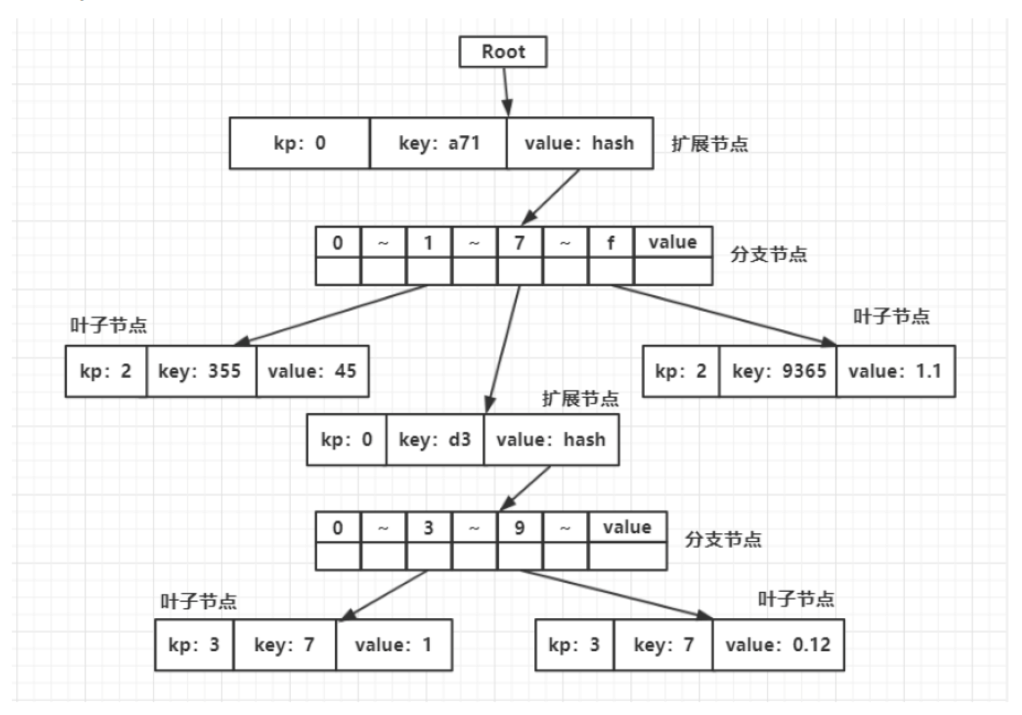



图 3-43　最终构建完成的 MPT 树及其各节点的数据情况

## 2.5.8　MPT 树如何体现默克尔树的验证特点

因为 MPT 树拥有默克尔树的特点，所以 MPT 树也具备默克尔树依靠节点哈希值来校验数据合法性的特点。那么 MPT 树是怎样利用节点的哈希值来实现数据校验的呢？

由 MPT 树节点持久化的特点可知，持久化时每个节点会生成对应的哈希值，而 MPT 树校验过程所使用的节点哈希值就是持久化时使用 RLP 数据生成的哈希值。回顾持久化的步骤，节点生成哈希值的顺序是从底部开始的，父节点哈希值的生成依赖孩子节点的哈希值，孩子节点的哈希值由其自身的 Key 和 Value 生成，最后生成树的根节点的哈希值。

因此，MPT 树在验证某个节点的合法性时也符合默克尔树从底部开始，逐级往上的验证过程，逐步生成父节点的哈希值，最后生成根节点的哈希值，然后和 Root 对比，判断它们是否相等，是则为合法节点，否则就是非法节点。

## 2.5.9　以太坊钱包地址存储余额的方式

以太坊区块 Header 结构体中“Root”变量的真实含义是，以太坊区块账户 MPT 树根节点的哈希值，区块账户 MPT 树中每个叶子节点的 Key 中存放的是以太坊钱包的地址值，叶子节点的 Value 对应的是以太坊的状态对象 stateObject。而状态对象 stateObject 中又含有账户 Account 对象，在 Account 对象中有一个指针变量 Balance，指向以太坊存放余额的内存地址，这也是以太坊的账户（Account）模型。

stateObject 对象和 Account 对象在代码中的定义分别如下：

```
type stateObject struct {
    address  common.Address
    addrHash common.Hash // 钱包地址的哈希变量形态
    data     Account  // Account 对象
    db       *StateDB
    dbErr error
    trie Trie // 首次访问，stateObject 还没有被纳入树节点中，它会是空值
    code // 只有当该账号是智能合约账号时，它才有值，对应的是合约的 bytecode
    ...
}
type Account struct {
    Nonce uint64
    Balance *big.Int
    Root common.Hash // 树根的哈希值
    CodeHash []byte
}
```

说明：

- Nonce，如果账户是用户钱包账户，Nonce 代表的是该账户发出当前交易时的交易序列号；如果账户是智能合约账户，Nonce 代表的是此账户创建的合约序号。
- Balance，该账户目前存放以太币余额的内存地址，请注意是以太币。
- Root，当前 MPT 树的根节点的哈希值。

- CodeHash，如果账户是用户钱包账户，该值为空，如果是智能合约账户，该值对应于当初发布智能合约代码的十六进制哈希值。

因为每个区块都对应一棵账户 MPT 树，就区块而言，它的账户 MPT 树中的账户数据都来源于被当前区块打包了的交易中，因为每笔交易中都存在着账户与账户之间的代币（Token）资产转移记录，区块打包了某笔交易，便会提取该交易中的账户资产信息作为账户 MPT 树的某个节点插入到树中。

## 2.5.10　余额查询的区块隔离性

我们知道，账户的 MPT 树的叶子节点依赖于当前区块打包了的交易数组，换句话说，账户 MPT 树记录的账户信息是基于区块的。由于以太坊节点同步的有效区块来源于公有区块链，因此节点之间存在同步区块的快慢情况，这种情况常会造成余额查询出错。

下面我们通过一个例子来加以说明。

假设公链的最新区块高度是 100，现在有两个以太坊节点 A 和 B，节点 A 同步区块到了高度 98，它把高度 98 打包了的交易中的账户信息逐个更新到<k,v>数据库中，而节点 B 同步区块到了高度 100，节点 B 也保存好了账户信息。

此时，假如一个以太坊节点 C 共有 8 个 ETH，且在之前发起了两笔交易，第一笔交易转账出去了 3 个 ETH，第二笔交易转账出去了 2 个 ETH，第一笔交易被区块 98 打包了，第二笔交易被区块 100 打包了。此时节点 D 调用以太坊的 RPC 接口查询节点 C 的以太坊 ETH 余额，被查询到的节点刚好是 B，那么节点 B 返回的将会是（8-3=5）的结果，而事实上节点 C 的真实余额是（8-3-2=3）个 ETH。

## 2.5.11　余额的查询顺序

虽然账户数据会被持久化到<k,v>数据库中，但是在进行账户余额查询时并不是直接到<k,v>数据库中查找，因为账户 MPT 树持久化的“键-值对”是一个巨量的<k,v>数据集，直接查询需要很长时间，为加快查询速度，以太坊在钱包地址中代币（Token）余额查询上设置了三级缓存机制。

我们再来看 stateObject 结构体，其中有一个 StateDB 类型的 db 对象指针，该 db 指针对象就存储了基于内存的缓存 Map。stateObject 和 StateDB 在代码中的定义如下:

```
type stateObject struct {
    address  common.Address
    addrHash common.Hash // 钱包地址的哈希变量形态
    data     Account  // Account 对象
    db       *StateDB
    ...
}
type StateDB struct {
    db Database //leveldb 对象
    trie Trie  // Trie 树的第二级缓存
    stateObjects map[common.Address]*stateObject // 第一级内存缓存
    stateObjectsDirty map[common.Address]struct{}
```

```
        ….
    }
```

余额的查找顺序是：

（1）第一级查找基于内存中的 stateObjects 对象，这里保留了近期活跃的账号信息。

（2）第二级查找基于内存中的 trie 树。

（3）第三级查找基于 leveldb，即<k,v>数据库层。

第一级和第二级查找都是基于内存的，第二级的 Trie 体现在代码上是一个接口，在 stateObject 中，trie 变量最终是一棵 MPT 树，它被用于在检验某一个钱包地址（address）的 stateObject 数据是否真的存在于某个区块中，其验证方式就是默克尔树的数据校验方式，这种设置优化了查找的整体时间复杂度。

## 2.5.12　UTXO 模型和 Account 模型的对比

根据前文对比特币 UTXO 模型和以太坊 Account 模型的介绍，可以得出以下几点结论：

（1）在计算方面，UTXO 本身并没有过多的复杂计算，且在链上的计算也不多，由于 Account 模型是图灵完备的，支持智能合约，它的运算大部分在链上，计算相对来说比较复杂。因为智能合约部分对应的是从 Solidity 编程到编译的整个过程，通过代码能够实现一切可计算问题，所以 Account 模型比 UTXO 模型更具备可编程性。

（2）在并发发起交易方面，UTXO 模型支持并发，因为它不受交易编号顺序的限制，所以可以无须考虑顺序而以批量方式发起交易。Account 模型因为存在 Nonce 交易序列号，所以它严谨地要求每笔交易的 Nonce 必须是递增的，也就是说，它的每笔交易都存在强关联性。

（3）在交易重放方面，UTXO 模型和 Account 模型都具备抵抗交易重发情况的功能。在 UTXO 模型中，因为每次交易的输入（Inputs）都和输出（Outputs）都存在从入到出的关系，如果一个相同的交易被重新发起，那么它所对应的输入在第一次的时候就已经被消费了，便会导致当前的交易失败，可以说是自身就带有抵抗交易重复的特点。相对来说，Account 模型的做法是采用强顺序性的交易序列号 Nonce 来抵抗交易重发问题。

（4）在存储方面，在 UTXO 模型中的交易记录存储在链上的区块中，这样时间一长，比特币的公链上，区块整体数据量会变得非常庞大。而 Account 模型存储在链上的只有 MPT 树的根节点的哈希值，实际的节点数据都持久地存放在每个节点本地的<k,v>数据库中。

（5）在余额查询效率方面，因为 UTXO 模型并没有直接存储某个钱包地址的资产余额，而是通过多个输入输出交易来记录资产的变化，从而导致在查询钱包地址中的资产余额时须先获取到所有相关的 UTXO 交易记录的列表，再汇总统计。而 Account 模型使用了三级缓存的形式，即使缓存中没有记录，其最终也会到<k,v>数据库中直接查询余额信息。

综上所述，Account 模型具备可编程性和灵活性，而 UTXO 则在简单业务和跨链上，有其独到和开创性的优势。

# 2.6 以太坊的版本演变

以太坊的发展主要体现在版本的演变上，类似于一个常规软件的升级流程，它的升级方式是先增加节点的代码再编译成对应版本的节点程序，然后发布，让其他节点同步更新升级。

以太坊每次升级都是为改善以太坊网络，或修复问题，或增强以太坊网络的性能，每个版本都有其各自的特点。

## 2.6.1 以太坊与 PoW 共识机制

以太坊源码在发展的过程中，其在不同阶段所使用的共识算法并不相同，下面分版本进行说明。

（1）Frontier（前沿）。这个版本是以太坊的基础，此时的以太坊具备了挖矿、交易及智能合约功能模块，但是没有供普通用户使用的图形化界面，仅适合开发者使用，所使用的共识算法是“PoW”。

（2）Homestead（家园）。这个版本的以太坊网络变得更加稳定，且具备了图形界面的钱包软件，所使用的共识算法还是“PoW”。

（3）Metropolis（大都会）。分为下面两个子版本：

- 拜占庭。发布了集合钱包功能以及合约发布等丰富功能的图形化界面软件“Mist”，同时也引入了很多新的技术，例如零知识证明和抽象账号等，使用的共识算法仍然是“PoW”。截至 2018 年 12 月 14 日，以太坊最新发布的版本是“Metropolis 大都会”的“拜占庭”。
- 君士坦丁堡。本计划使用混合共识算法“PoW+PoS”，但最终依然是“PoW”，为“宁静”做铺垫。

（4）Serenity（宁静）。该版本将把以太坊的共识算法全部换成基于“PoS”的变种算法“Casper 投注共识”，它属于“PoS”系列。

由上可知，在以太坊发展的过程中，其共识算法在不同的阶段经历了从“PoW”共识、“PoW+PoS”共识到“PoS”共识，可以说，以太坊的共识算法是从“PoW”开始的。

## 2.6.2 君士坦丁堡

作为以太坊大都会版本的子版本，君士坦丁堡的升级已经确定在主网区块高度 7080000 的时候激活，北京时间大致在 2019 年 1 月 14 日到 18 日之间。

因为本次升级获得了所有公网节点的认同，意味着不会出现分叉币，同时也不会影响到已有以太坊地址的 ETH 代币。作为以太坊的公网节点需要准时同步升级节点的程序，以确保节点在新的链上继续挖矿生产区块。

下面我们来介绍君士坦丁堡版本的一些主要特性：

（1）共识机制，依然是“PoW”。

（2）加入了下面的 EIP（Ethereum Improvement Proposal），EIP 中文全称是以太坊改进建议。

- EIP145: 出自以太坊开发人员 Alex Beregszaszi 和 Pawel Bylica。主要引进了一种叫作“位移”（Bitwise Shifting）的运算符。以太坊虚拟机（EVM）之前缺少这种运算符，只支持其他逻辑和算术运算符，“位移”运算符只能通过逻辑和算术运算符实现，现在通过原生支持的“位移”运算符能优化智能合约类 DApp 的 Gas（燃料）消耗，因为 Gas 的消耗与字节数据量的多少有关。
- EIP1014: 由以太坊创始人 Vitalik Buterin 亲自提出。新增了一个合约创建函数 CREATE2，提供了一种可以提前预测合约地址的合约创建方法，该升级能更好地支持基于状态通道或者链下交易的扩容解决方案，即现在主流的 Layer2 方案。
- EIP1052: 出自以太坊核心开发人员 Nick Johnson 和 Pawel Bylica。引入了一个新的操作码，允许直接返回合约字节码的 keccak256 哈希值，该升级能有效地减少以太坊网络对于大型智能合约的运算量，尤其是在只需要智能合约的哈希值的时候。
- EIP1234: 该升级主要是将现有的区块奖励由 3ETH 减少到 2ETH，减少了 33%，同时将难度炸弹（Difficulty Bomb）推迟了 12 个月。
- EIP1283: 该升级通过更改 SSTORE 操作码优化智能合约网络存储的 Gas 值，减少了和智能合约运行量不匹配的 Gas 消耗。

总的来说，此次升级可以概括为下面的 3 点：

（1）从以太坊底层虚拟机到智能合约的一系列内容，提高了整个以太坊网络的性能。

（2）位移运算符、新的虚拟机操作码、优化合约网络存储的 Gas 值，使得虚拟机运算合约代码速度更快，运算量更少，最终消耗合约调用者的 Gas（燃料）更少，对开发者更加友好了。

（3）难度炸弹延缓一年，区块奖励从 3 减少到 2，导致节点中矿工的实际收益减少了 1/3，直接关系到矿工的利益。

# 2.7　以太坊 Ghost 协议

“Ghost 协议”的全称是“Greedy Heaviest-Observed Sub-Tree protocol”，中文称为贪婪子树协议，又称幽灵协议，它属于主链选择协议范畴。

首先，比特币公链是根据最长链规则来解决区块链分叉问题的，但并不是所有的区块链公链解决分叉问题都是使用最长链规则，以太坊就不是。

以太坊解决区块链分叉问题目前使用的是 Ghost 协议，而 Ghost 协议的真实作用是用来进行主链选择。不同于比特币的最长链规则，以太坊在选择最长链时不以哪条链区块连续最长为标准，而是将分叉区块也考虑了进去，选择出一条包含了分叉区块在内区块数目最多的链作为最长链，如图 2-44 所示。

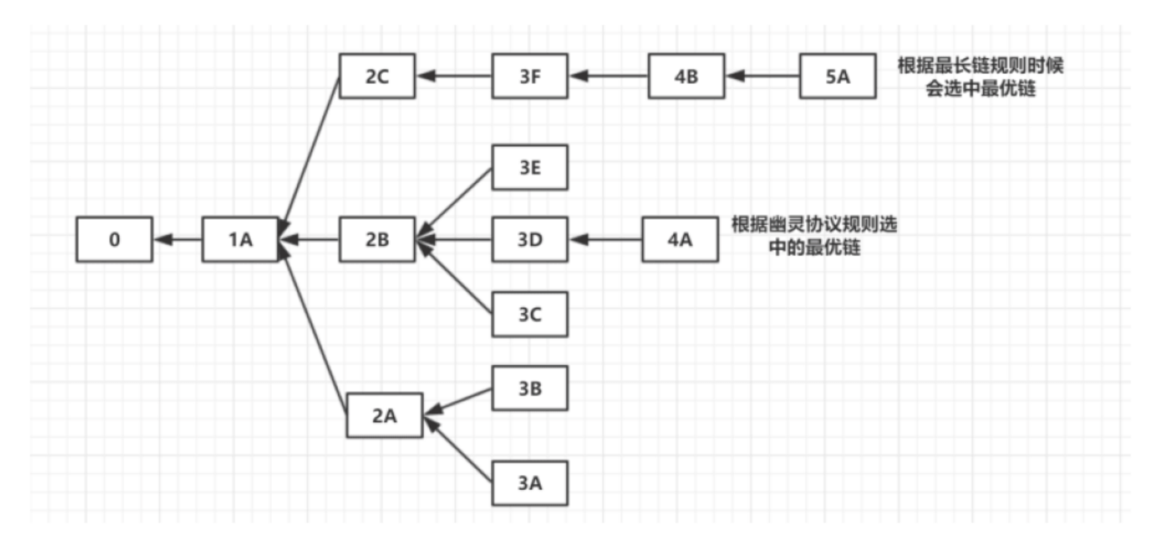



图 2-44 Ghost 协议和最长链规则选中的不同链

请看图 2-44 的分叉情况，在比特币公链中最终胜出的链是 0<-1A<-2C<-3F<- 4B <- 5A，这是一条由最长链规则选择的链。而在以太坊公链中，根据幽灵协议最终胜出的链是 0<-1A<-2B<-3D<-4A。原因是在图 2-44 的分叉情况中，幽灵协议把分叉区块也考虑进去了，统计总的区块数，发现在包含了区块 0、1A、2B、3E、3D、3C、4A 的链是含有区块数最多的。因此该链胜出，这就是幽灵协议选择最优链的机制。

此外，对于在最长链中被包含进去的造成链分叉的区块，例如图 2-44 中的 3E 和 3C，Ghost 协议对它们也有一套对应的处理机制，这些区块会根据规则被处理为：

（1）孤块。完全没用的区块，挖出这个区块的矿工没有任何收益。比特币链中的分叉区块都是孤块。

（2）叔块。被一定范围内的后续子区块打包收纳的区块，挖出叔块的矿工会按照一定算法给予收益。

综上所述，我们知道，Ghost 协议在以太坊中主要起到以下两点作用：

（1）选择出最优链。

（2）对最优链中分叉块进行处理。

# 2.8 Casper：PoS 的变种共识机制

前面谈到，以太坊 Serenity（宁静）版本将会把共识机制完全切换成“PoS”的共识机制，这个共识机制还有另外一个名称——“Casper 投注共识”。以太坊的“Casper 投注共识”属于“PoS”共识机制范畴，它是在“PoS”股权证明思想上拓展衍生出的一种股权证明机制。因为“Casper”版本还没有完全公布，笔者也只能从现有的资料中归纳出它的一些特点。

“Casper 投注共识”增加了惩罚机制，并基于“PoS”的思想在记账节点中选取验证人，验证人对应的就是股权拥有者，投注是验证人所拥有的动作，且能够投注的角色只能是“验证人”。可

以将这类角色理解为新一代以太坊矿工，因为投注如果获胜是会有收益的，相当于挖矿收益。

投注指的是在“Casper 共识机制”中，验证人要拿出保证金的一部分对它认为的大概率胜出区块进行下注，类似于赌博，投注所能产生的结果是：

（1）赌对了，可以拿回保证金外加区块中的交易费用，也许还会有一些新发的货币。

（2）下注太慢没有迅速达成一致，能拿回部分下注金，相当于损失了一些下注金。

（3）数个回合之后下注的结果出来，那些选错了的验证人会输掉下注金。

（4）验证人过于显著地改变下注，例如先赌某个区块有很高的概率胜出，然后又改赌另外一个区块有很高的概率胜出，将会被惩罚。

## 2.8.1　如何成为验证人

想成为验证人，需要交保证金进行申请，同时也可以在进入后选择退出，加入和退出都将会成为以太坊网络中的一种特殊交易类型，目前最常见的交易就是转账 ETH 代币。也就是说，到时候可能要调用一定的以太坊接口来申请成为验证人。保证金很有可能就是以太坊 ETH 代币，它将会被用来投注，或因被以太坊的惩罚而没收掉投注金。

目前 Casper 的验证人逻辑通过一个名称为 Casper 的合约来实现，该合约提供投注、加入、取款和获取共识信息等一系列功能，因此通过简单地调用 Casper 合约就能提交投注或者进行其他操作。Casper 合约的内部状态如图 2-45 所示。

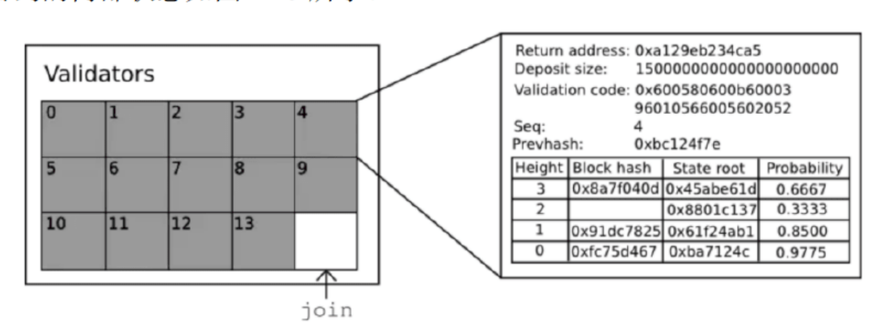



图 2-45　Casper 合约验证人的数据字段组成

从图 2-45 可以看到，这个合约记录了当前验证人的信息，每位验证人有 6 项，分别说明如下：

- Return address，验证人保证金的返还地址（钱包地址）。
- Deposit size，当前验证人保证金的数量（注意验证人的投注会使这个值增加或减少）。
- Validation code，验证人的验证代码。
- Seq，最近一次投注的序号。
- Prevhash，最近一次投注的哈希值。
- 验证人每次投注的表格。

## 2.8.2　验证人如何获取保证金

目前验证人获取保证金的方式，或者说获取代币的方式，主要是基于“PoS”共识机制，即可

以通过转让、交易的方式来获取。

如果是早期版本则是基于 PoW 挖矿获取，如果涉及网络升级，还要考虑兼容旧节点的情况。

### 2.8.3 候选区块的产生

验证人要投注的对象是区块，那么在“Casper 投注机制”中区块将由谁产生？毕竟只有区块被产生了才能有投注的动作。

答案是，区块将由验证人出块。出块是一个独立于其他所有事件而发生的过程：验证人负责收集交易，当轮到它们的出块时间时，它们就制造一个区块，然后签名发送到节点网络上去。

轮流出块的规则也是由“Casper”提供的。

### 2.8.4 胜出区块的判断

等所有验证人都在限定的时间内投注完了，在所有压了注的区块中哪个将会胜出呢？

区块胜出的规则是这样的：当验证人中的绝大多数，即满足协议定义阈值的一群验证人的总保证金比例达到 67%到 90%之间的某个百分比，并以非常高的占比率下注某个区块胜出的时候，此区块便会胜出。

不难看出，“Casper”的投注方式存在验证人联盟共同投注某个区块使之胜出的非公平问题。对于这个问题，目前以太坊还没有很好的解决方案。

## 2.9 智能合约

### 2.9.1 简介与作用

我们生活中所认识的合约又称合同，是基于文字制定的条款，例如劳工合同。

在以太坊中，智能合约也可以看作是一份合同，它的表现方式是：使用规定的计算机语言编程，然后编写一份代表合同的代码文件，再经过编译，变成可被执行的计算机字节码。

可见，以太坊的智能合约也可以理解为一份代码文件，例如用 C++语言编写的是.cpp 文件，用 Java 编写的是.java 文件。

目前以太坊智能合约的编程语言是 Solidity，采用 Solidity 语言就可以编写出各种各样的智能合约，然后将其部署到以太坊上，部署的详细流程是：

（1）编写好智能合约代码文件。

（2）经过 Solidity 编译器，将代码文件编译成十六进制码。

（3）将编译好的十六进制码，以以太坊交易的形式发送到以太坊网络上。

（4）以太坊识别出是部署合约的交易，校验后，存储起来。

（5）待合约被链下请求调用的时候，以太坊智能合约虚拟机（EVM）将编写好的智能合约代码文件编译成二进制码，并加载运行。

从部署到被运用的整个流程，产生了所谓的基于智能合约的 DApp 应用。

请看下面智能合约的例子：

```
pragma solidity ^0.4.17;
contract MathUtil {
  function add(uint a,uint b) pure public returns (uint) {
    return (a+b);
  }
}
```

很明显，这是一个简单的加法操作，但这也是一份智能合约，只不过是一份简单的智能合约。

注意，每一份被部署到以太坊上的智能合约都有一个唯一标识的哈希地址值，这个哈希地址值既代表用户的以太坊账户地址，又唯一标识了一份智能合约。

我们知道，代码在编译成可执行的字节码之后是可以被调用执行的，同样地，所有被编译布署到以太坊中的智能合约也可以被调用，也就是上面的加法智能合约是可以被调用的。注意，这里的调用指的是合约里所编写的函数可以被以各种方式调用。可以定义私有函数，供智能合约调用；也可以添加 Owner 权限（只能是 Owner），由合约发布者调用或公共调用。

对于能够被公共调用的智能合约函数，其所面向的最为广大的调用者就是所有人，你可以调用，他、我也可以调用。怎样调用呢？可以通过以太坊提供的 RPC 接口。当然，以太坊也提供了传统的 RESTful API 的调用方式，也就是我们可以将调用智能合约的函数理解为调用服务端接口。

下面我们来理清一些关系：

- 智能合约→被布署到以太坊节点上→调用时被以太坊虚拟机编译并加载。
- 以太坊节点→被布署在不同的服务器上→节点们共同维护以太坊公链。
- 调用者→调用以太坊节点的接口→访问某个智能合约→获得结果。

我们知道，节点网络分为公链节点网络和私链节点网络，在不同类型的节点网络中部署的智能合约，其访问域是不同的，私有节点网络部署的智能合约只能在访问私有节点网络时才能访问到这个合约，而公有节点部署的智能合约，所有人都可以调用。

如图 3-46 所示是访问私有节点网络智能合约的模型图。

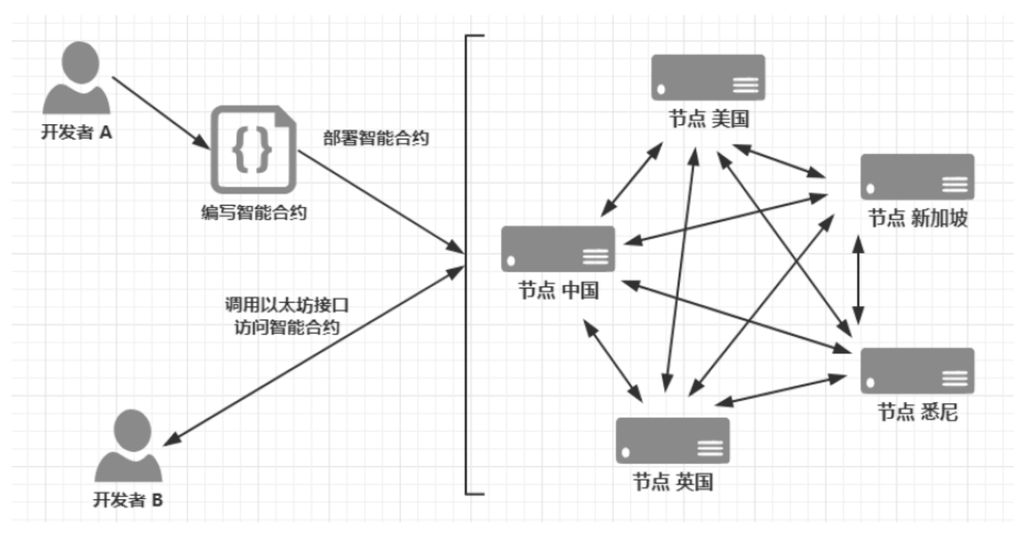



图 3-46　访问私有节点网络智能合约的模型图

部署在以太坊网络上的智能合约就像部署了一个服务端程序，我们通过调用在智能合约中编写的函数可以实现各种应用，这正是以太坊智能合约的作用。例如，ERC20 代币的标准智能合约代码中就有一个转账函数，而所有的 ERC20 标准的代币合约，它们的转账就是通过调用这个函数实现的。也就是说，ERC20 代币转账就是基于智能合约的，ERC20 的代币有很多种，每一种代币对应一份智能合约。我们发布 ERC20 代币到链上，其本质就是发布一份智能合约。

## 2.9.2 合约标准

我们知道，动物是生物的一个种类，在动物的大范围下又分人类、猫类、鱼类等。类似地，以太坊智能合约也是一个对合约的统称，在此合约下，又有特别针对一类合约的标准，例如标准的代币合约标准——ERC20 代币标准、ERC721 标准等，如图 2-47 所示。




图 2-47 以太坊智能合约的分类

本书我们主要介绍两种广泛使用且代表性比较强的合约标准：ERC20 与 ERC721。

### 1. ERC20 标准

我们首先来认识 ERC20 标准。

ERC 的全称是“Ethereum Request for Comments”，中文含义为“以太坊征询意见”。后缀添加的数字（例如 20、223 等）是版本号。

ERC20 标准的官方解析链接：

https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

ERC20 标准的诞生起因于以太坊的应用本质，由于以太坊目前几乎都是被应用于虚拟货币中，包括以太坊本身也有代表它自己的虚拟货币：ETH。因此，几乎所有使用以太坊智能合约在以太坊上部署的合约都是代表虚拟货币的智能合约。

作为货币，它自然有货币的属性，例如货币名称、货币发行量及货币的唯一标识等。为什么名称不是唯一标识？这是因为以太坊限定了进行唯一标识的只有哈希值，所以虚拟货币的名称是允许重复的，比如两份不同的 ERC20 标准合约，它们代码中的 name 或 symbol 变量都可设置为 ETH2。货币除拥有上面的属性外，还必须允许用户查询余额和转账，这些都是货币所拥有的基本特点。

对于代表虚拟货币的智能合约来说，为方便虚拟货币的发布，催生了 ERC20 标准，该标准现在已经被专门用来发布虚拟货币，标准中包括了成员变量、函数和事件等，以方便开发者调用。

ERC20 标准是一种软性强制的标准，因而并不是发布虚拟货币都必须遵照这个标准，这标准里面的一些属性和函数，开发者可以遵循也可以自己创新。但是，请注意，目前很多的以太坊钱包软件在设定进行代币转账时，默认使用 ERC20 标准的函数名称和传参类型。所以，如果你的代币

不遵循该标准发行，就有可能导致钱包软件转账失败。

下面我们根据官方文档对 ERC20 标准的成员变量、函数和事件进行讲解。

（1）标准的成员变量

ERC20 标准规定了智能合约在使用 Solidity 语言编程时可以通过下述形式来定义成员变量：

- string public name;
- string public symbol;
- uint8 public decimals;
- uint256 public totalSupply;

说明：

① string 用来定义 name 和 symbol 为字符串类型的变量，uint8 表示 decimals 是 8 位（bit）的无符号整型数字，uint256 定义变量 totalSupply 是 256 位（bit）的无符号整型数字。无符号的整型数字可取的正数范围变大了，其最小值是 0，但不能取负数。

② name 一般表示当前代币的名称，例如 My First Token。

③ symbol 表示当前代币的符号，代表的是一种简称，例如可以取 name 的 3 个首字母来设置，My First Token 的 3 个首字母是 MFT。

对于 symbol，请注意以下两点：

- 我们一般口头上说一个代币的时候，说的都是 symbol 符号。例如，LRC 就是一个 symbol 符号。
- symbol 不能唯一标识一个代币。symbol 是可以重复的，只有代币的合约地址才能唯一标识代币，所以不要以 symbol 来唯一标识一个代币。

一般来说，name 和 symbol 都可以任意设置，也可以设置为同一个字符串，但要正规地表示一个代币，还是要进行妥善设置，因为这些合约的代码都能在“区块链浏览器”被搜索并且浏览的。

④ decimals 表示将代币单位精确到小数点后多少位，比如总量初始化为 1000，decimals 为 1（即代币单位精确到小数点后 1 位，也就是 0.1），则实际是 100 个代币（$100\times10^1 = 1000$，即有 1000 个 0.1），此时如果你要从钱包软件中向别人发送 1 个代币，在钱包里不能写 1，而是要写 10，因为写 1 表示发送 0.1 个代币（因为精确到 0.1），通过交易可以查看到实际发送的就是 0.1，所以如果你要发行 1000 个代币，那么在智能合约中的初始设置应该是 $total = 1000\times10^{decimals}$（即要乘上 10 的 decimals 次方），发行的数量需要相对代币小数点后的位数来设置。例如，如果精确到小数点后的位数是 0，而你要发行 1000 个代币，那么发行数量的值是 1000，因为代表单位精确到 1。但是，如果代币单位精确到小数点后的位数是 18 位，你要发行 1000 个代币，那么发行数量的值就是 1000000000000000000000（1000 后面加上 18 个 0），因为代币单位精确到小数点后 18 位。

⑤ totalSupply 代表当前代币的总发行量，留意到它对应的是 uint256 整型数字，即 256 位的无符号整型数字，而不是 8 位，就知道这个数字表示的范围是很大的。假设 decimals 是 18，然后我们发行量是 100 亿个代币，那么此时 totalSupply 的真实数值是 $totalSupply = 100\times10^{18}$。这个数字非常之大，一般整型会溢出，所以要按照标准的规则来定义好你的变量，以避免出现数据溢出的错误。

以上我们介绍了 4 个标准的成员变量，那么是不是一定要按照标准必须使用这 4 个变量呢？不是的，请记住，智能合约中的代码可以不按照标准写，例如代币的名称，标准中要求使用 name 来表示，但是你想换个变量来表示，比如换成 tokenName，那么需要在合约中编写特定的函数，以便合约调用者可以访问到这个 tokenName 变量。

具体见下面 ERC20 函数的说明。

（2）标准的函数

ERC20 标准规定了智能合约须具备并实现下面的函数及事件（Event）：

```
    contract ERC20 {
        function totalSupply() constant returns (uint256 totalSupply);
        function balanceOf(address _owner) constant returns (uint256 balance);
        function transfer(address _to, uint256 _value) returns (bool success);
        function transferFrom(address _from, address _to, uint256 _value) returns
(bool success);
        function approve(address _spender, uint256 _value) returns (bool success);
        function allowance(address _owner,address _spender) constant returns
    (uint256 ret);

        event Transfer(address indexed _from, address indexed _to, uint256 _value);
        event Approval(address indexed _owner, address indexed _spender, uint256
_value);
    }
```

以上 ERC20 标准中的各个函数都要求使用代码来实现，具体怎么实现，标准并不关心，只需要返回每个函数所规定的参数类型即可。例如，balanceOf 函数的功能是查询钱包地址的代币余额，只要结果返回余额的值即可。这种情况就像 Java 语言中的接口，定义好接口，具体的实现，Java 并不关心。

下面我们对上述标准中的各个函数分别进行说明。

① 返回代币发行量的函数 totalSupply

```
    totalSupply() constant returns (uint256 totalSupply)
```

这个函数要求返回当前代币的总发行量，返回的值就是 totalSupply 的数值。注意，如果你不明确地在智能合约中写出返回 totalSupply 的函数，但是定义了 totalSupply 变量，那么 EVM 虚拟机在编译的时候会自动帮你加上返回 totalSupply 的函数。例如，下面的这个函数是在定义了 totalSupply 变量但没有明确写出 totalSupply 这个函数时 EVM 自动加上的。

```
    function totalSupply() constant returns (uint256 totalSupply){
         return 1000000000000000000000  // 已经自动乘上了 decimals 的格式
    }
```

② 返回代币余额的函数 balanceOf

```
    balanceOf(address _owner) constant returns (uint256 balance)
```

balanceOf 的作用是返回一个钱包地址所拥有当前代币的余额，供查询余额所用，只需要传入一个钱包地址，address 类型代表的就是地址类型，最后返回的是代币的余额，其结果也是乘上了 decimals 后的数字格式。

③ 转账函数 transfer

```
transfer(address _to, uint256 _value) returns (bool success)
```

transfer 的作用，顾名思义就是转移，即用于转移代币的转账函数。入参分别是要接收代币的以太坊地址_to，以及要转多少的数值_value。你可能想到了，为什么没有 from？从哪个地址转出呢？答案是这个函数内部的实现一般都是把下面的两种地址角色作为默认的转账地址：

- 当前调用这个转账函数的地址 msg.sender ，它是函数代码中的一个变量。
- 合约创建时所设置的最初的收币地址。

关于上面的第二点，这里举例做一个说明。

假设钱包地址 XXX 是合约 A 此刻 transfer 的调用者，这时 A 的调用者 msg.sender 就是 XXX，然后在智能合约代码里的 transfer 函数实现的时候要写明从地址 YYY 中转出，如下代码所示：

```
function transfer(address _to, uint256 _value) returns (bool success) {
    ...
    balanceOf [ YYY ] -= _value;  // 注意这行的 YYY 作为默认转出地址
    balanceOf [ _to ] += _value;
    ...
}
```

转账相关的函数还有 transferFrom，它和 transfer 一样，也用于实现转账的功能。

```
transferFrom(address _from, address _to, uint256 _value) returns (bool
success);
```

不同的地方在于转账的形式：transferFrom 是从某个钱包地址_from 向_to 转账，_from 是传参进来的，这就意味着我们可以设置任何钱包地址为转出地址。这里要注意的是，使用这个转账函数的前提是必须获得授权。

④ 授权函数 approve

```
approve(address _spender, uint256 _value) returns (bool success);
```

approve 就是授权函数。在使用 transferFrom 前要对传入 transferFrom 中的 from 地址进行它所在当前代币的授权值的判断，只有这个授权值满足了给定值才能使用 transferFrom 函数。

那么，为什么要授权呢？我们通过一个例子来了解一下原因，这和你委托你的一个朋友去帮你转账给另外一个人的情况是一样的。例如，A 叫 B 帮 A 转账人民币 100 元给 C，这个时候由于 B 只是一个帮忙转账的人，它是没有 A 的银行卡和密码的，只有在得到 A 的授权后才能操作，而且授权也是有一个数值的，例如 100 元。那么 A 就先向银行 D 授权自己的转账权限给 B，允许 B 代替 A 转账给 C 共 100 元。

以上就是 approve 的授权流程，首先合约 A 的调用者 msg.sender 在合约 A 中授权给_spender，允许_spender 能够代替自己转账_value 个数值的代币。此后，_spender 就能在合约 A 中调用 transferFrom 从_from 中转账_value 个代币给_to，注意这时的_from 就是当初调用 approve 的 msg.sender。为了加深理解 approve 和 transferFrom，下面再提供一个流程图，如图 2-48 所示。

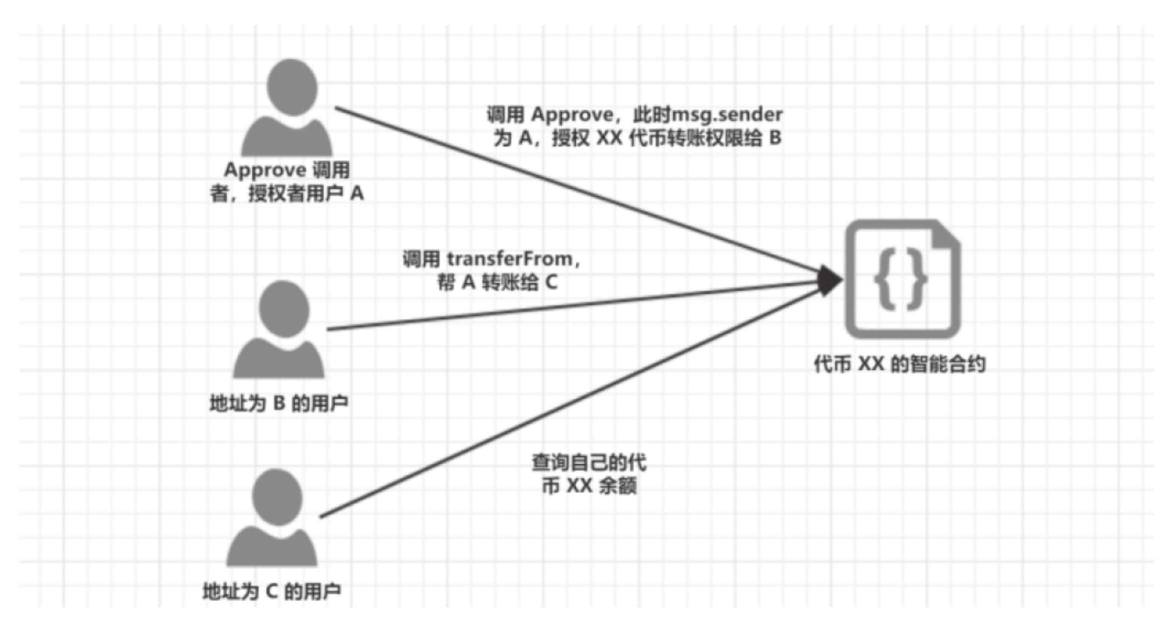



图 2-48 使用 transferFrom 转账的流程

一般来说，transferFrom 的内部实现都会对授权值进行判断，当然，你也可以不判断，但这就不是标准的做法了。如果做了判断，发现当前调用 transferFrom 的 msg.sender 还没有授权值，就会报错。这种错误统称为合约层的非编译时错误，只能通过查看智能合约代码来分析错误原因。下面是 approve 和 transferFrom 判断授权值的实现代码示例：

```
    function transferFrom(address _from, address _to, uint256 _value) public
returns (bool success)
    {
        uint256 allowance = allowed[_from][msg.sender];  // 取出数值
        ...    // 进行数值判断，成功后额度数值减去转出部分等
        return true;
    }

    function approve(address _spender, uint256 _value) public returns (bool success)
    {
         allowed[msg.sender][_spender] = _value;   // 进行授权值设置
         ...
         return true;
    }
```

⑤ 授权额度查看函数 allowance

```
    allowance(address  _owner,address_spender) constant returns  (uint256
ret);
```

allowance 所对应的是 approve 所授权的额度查询，它会返回_owner 地址到当前代币合约 XX 中，方便查询_owner 给_spender 授权了多少个 XX 代币的数值，也是我们在开发过程中经常使用的函数。

（3）标准的事件（Event）

上面我们介绍的是 ERC20 标准的函数，其实 ERC 标准还有两个 Event 事件类型，下面对这两个事件进行详细介绍。事件是 Solidity 编程语言语法中的一类特性，其作用是当该事件的代码被

EVM 虚拟机调用触发时能够以消息方式响应调用者前端，类似于 Java 语言中的回调函数（callback）。

也就是说，我们可以自己在代码中定义想要的事件（Event）。在 ERC20 标准中，规定了在编写转账、授权函数代码时，必须在成功转账后触发转账事件。我们首先介绍转账的事件 event。

```
event  Transfer(address indexed _from, address indexed _to, uint256 _value);
```

Transfer 事件需要在 transfer 和 transferFrom 函数内触发。如果你留意这两个函数的返回值，就会发现返回的都是 bool（布尔）类型。但是，请注意，在真实调用的时候，并不是直接通过 RPC 接口调用这两个函数，而是通过以太坊的交易接口来调用智能合约的转账函数。

在调用以太坊的交易接口时，以太坊会返回一个 TxHash 值，也就是交易的哈希凭据值。此时客户端也就是调用者还不能马上知道交易结果，之前的内容提到过，以太坊的交易需要矿工打包到区块中，所以需要等待交易被矿工打包到区块后才能得知最终的结果。

等待时间的长短是不确定的，在这种情况下就需要一个 event（事件）来通知，待交易被矿工打包到区块后，EVM 虚拟机会执行智能合约的转账函数，最后触发 event（事件），随后客户端就能在监听代码中处理最终的结果。下面是 web3.js 的一个例子。

```
// 实例化代币的智能合约对象
var contract = new web3.eth.Contract(TokenABI,TokenAddress);
// 发起转账，txHash 是能够马上被返回的
var txHash = contract.sendCoin.sendTransaction(To, 100, {from:From})
// 获取事件对象
var myEvent = contract.Transfer();
// 监听事件，监听到事件后会执行回调函数
myEvent.watch(function(err, result) {
    if (!err) {
        console.log(result);
    } else {
        console.log(err);
    }
    myEvent.stopWatching();
});
```

此外，在 event 事件中，存在一个有着特殊意义的变量关键字，即“indexed”。在以太坊的事件机制中，对于成功触发的事件，以太坊会对事件进行数据层面的存储，方便开发者用筛选器（Filter）查找，所存储事件的数据区域对应的术语是“Event Log”（事件日志）。“Event Log”分两部分，分别是：

- Topic 部分（主题部分）。在智能合约函数中凡是被定义为“indexed”类型的参数值都会被保存到这个主题部分。
- Data 部分（数据部分）。没有被定义为“indexed”类型的参数值会被保存到这个数据部分。

一个 event（事件）中最多可以对 3 个参数添加 indexed 属性标签，添加了 indexed 的参数值会存到日志结构的 Topic 部分，便于快速查找，而未加 indexed 的参数值会被保存在 Data 部分，成为原始日志。需要注意的是，如果添加 indexed 属性的是数组类型（包括 string 和 bytes），那么只会在 Topic 部分存储对应数据的 web3.sha3 哈希值，将不会再保存原始数据。因为 Topic 部分是用于快速查找的，不能保存任意长度的数据，所以通过 Topic 部分实际保存的是数组这种非固定长度数

据的哈希值。如图 2-49 所示是在“区块链浏览器”中查询某笔交易记录的“Event Logs”（事件日志）时得到的结果。




图 2-49　区块链浏览器的 Event Logs

和 transfer 事件一样，ERC20 标准在代币授权成功后也有一个对应的授权事件触发。

```
    event  Approval(address indexed _owner, address indexed _spender, uint256
_value);
```

以太坊结合 Solidity 语言中事件机制，它的最为重要的作用就是能够给调用者客户端一个回调功能，即异步回调，这样才能处理交易或授权的结果。试想一下，我们转了一笔账，却不知道交易的结果是怎样的，转账的时候只有一个交易哈希值拿到手，要想知道结果，只能不断地使用这个哈希值去调用以太坊的接口进行查询，或者手动去区块链浏览器中查询。这样无论是从编写代码层面还是用户在应用层面的体验来说都不那么友好，特别是批量交易的应用场景，所以事件的回调机制在一定程度上解决了这个问题。

捕获交易结果除了使用事件回调监听形式，还可通过遍历区块解析其过程来达到目的，这个方法我们会在后续章节中介绍，并给出代码实现。

### 2. ERC721 标准

以上我们认识了专门为代币而设置的 ERC20 标准，但是在现实的开发中，除了使用智能合约来发布代币之外，更多的是实现和生活中实业相结合的智能合约应用。

想象一下，现实生活中，人与物理资产的对应关系都是一对一的，例如你买了一辆车，这个车有一个唯一的车牌号，且所有权归你，这就是一对一的关系。如果要把这种关系使用智能合约映射到区块链上，就需要制定一类合约标准来专门规范这种一对一的资产关系。

于是，ERC721 标准诞生了。ERC721 的官方解释是“Non-Fungible Tokens”，简写为 NFTs，翻译为非同质代币，或不可替换的代表。

什么是非同质代币呢？关于这个名词的解析，我们可以从 ERC20 和 ERC721 的区别来进行。ERC20 标准是专门为发布虚拟货币（即代币）制定的，货币的发行有发行量，例如共 10 万枚代币，

这些代币都是一样的，没有唯一的标识，假设这个虚拟货币的 symbol 符号是 XXX，ERC 标准就把这 10 万枚代币统称为 XXX 币。而在 ERC721 标准中，它把个体唯一化了，同样是 10 万枚代币，假设使用 ERC721 标准发布这份智能合约，那么这 10 万枚代币的每一枚都会单独有一个 ID，也就是说，10 万枚中每一枚都各自有唯一的标识，彼此互不相同，单位为 1，且无法再分割。

以上就是 ERC20 和 ERC721 最为核心的区别，主要表现在合约所表示的物质的个体化与一类化方面。ERC721 的这个特点——所表示的物质（代币）独一无二，使其更具有价值。该标准很好地映射了现实生活中一对一的关系。例如，生活中每辆车的车牌号是独一无二的，我们所养的宠物的基因也是独一无二的，等等。

2017 年，有一款基于 ERC721 标准开发的 DApp 游戏——CryptoKitties（加密猫），又称以太猫。这款游戏中的猫对象就是独一无二的，每只猫相当于一个代币，都拥有一个唯一标识的 ID。

如果把物理世界的资产与区块链智能合约结合起来看，ERC721 合约显然拥有更广泛的应用场景。但在 DApp 开发中，究竟使用哪一种合约标准，要根据项目的需要来决定。

（1）标准的成员变量

ERC721 标准所规范的成员变量和 ERC20 标准的基本一样，但是 ERC721 成员变量可以不需要 decimal 变量。

name 依然代表当前智能合约的名称，symbol 依然是符号简称，totalSupply 代表当前有多少个唯一代币（Token）。

此外，除了标准限定的成员变量，为了达到 ERC721 的要求，一般还需要一些 map 数据结构的变量来辅助实现代币的拥有者和当前代币一一对应的关系。

（2）标准的函数

合约的函数和事件也和 ERC20 的大部分一样，如下所示：

```
    contract ERC721 {
        // Required methods
        function totalSupply() public view returns (uint256 total);
        function balanceOf(address _owner) public view returns (uint256 balance);
        function ownerOf(uint256 _tokenId) external view returns (address owner);
        function approve(address _to, uint256 _tokenId) external;
        function transfer(address _to, uint256 _tokenId) external;
        function transferFrom(address _from, address _to, uint256 _tokenId)
external;

       // ERC-165 Compatibility (https://github.com/ethereum/EIPs/issues/165)
        function supportsInterface(bytes4 _interfaceID) external view returns
        (bool);

        // Events
        event Transfer(address from, address to, uint256 tokenId);
        event Approval(address owner, address approved, uint256 tokenId);

        // Optional 可选实现
        function name() public view returns (string name);
        function symbol() public view returns (string symbol);
        function tokensOfOwner(address _owner) external view returns (uint256[]
        tokenIds);
```

```
        function tokenMetadata(uint256 _tokenId, string _preferredTransport)
        public view returns (string infoUrl);

    }
```

相比 ERC20 标准，在必须实现的函数中，ERC21 标准多了 ownerOf 函数与 supportsInterface 函数。

```
    function ownerOf(uint256 _tokenId) external view returns (address owner);
```

ownerOf 的入参只有一个 tokenId，作用是返回当前拥有这个 tokenId 的代币的拥有者的地址。

```
    function supportsInterface(bytes4 _interfaceID) external view returns (bool);
```

supportsInterface 是 ERC165 标准的函数，ERC721 标准也会用到这个函数。ERC165 标准的原型是：

```
    interface ERC165 {
        // @notice Query if a contract implements an interface
        // @param interfaceID The interface identifier, as specified in ERC-165
        // @dev Interface identification is specified in ERC-165. This function
        //  uses less than 30,000 gas.
        // @return 'true' if the contract implements 'interfaceID' and
        //  'interfaceID' is not 0xffffffff, 'false' otherwise
        function supportsInterface(bytes4 interfaceID) external view returns
        (bool);
    }
```

根据官方对 ERC165 标准的注释，该标准主要的作用是用来检测当前智能合约实现了哪些接口，可根据 interfaceID 来查询接口 ID，存在就返回 true，否则返回 false。该标准函数还会消耗 Gas（燃料），至少消耗 30000 Gas。

下面举例加以说明。

假设一个 ERC721 智能合约里面有一个函数的名称是“getName”，先计算出该函数的 bytes4 类型的 ID：

```
    bytes4 constant InterfaceSignature_ERC721 = bytes4(keccak256(getName()'))
```

supportsInterface 的内部实现如下：

```

    function supportsInterface(bytes4 _interfaceID) external view returns (bool){
        return _interfaceID == InterfaceSignature_ERC721;
    }
```

当_interfaceID 传参后，直接进行 bytes4 类型的等值判断。

上面的 supportsInterface 函数是必须实现的。此外，ERC165 标准在可选的实现函数中还有一个看起来比较难理解的函数——tokenMetadata。

```
    function tokenMetadata(uint256 _tokenId, string _preferredTransport) public
view returns (string infoUrl);
```

tokenMetadata 的作用主要是返回代币的元数据（Metadata），内部返回的是我们自定义的一个字符串。元数据是什么意思呢？就是基础信息，例如合约里的 name 和 symbol 就是基础数据，

就好像一个人有名字、年龄和性别一样，这个函数的作用就是返回这些基础数据。一般来说，tokenMetadata 可以用来返回当前智能合约的创建日期是什么时候、名称是什么等这些基础数据，然后将这些基础数据拼接成一个字符串返回。

在事件机制方面，ERC721 和 ERC20 是完全一样的，这里就不再赘述了。

关于智能合约的标准，除了 ERC20 和 ERC721，还有很多，但并不常用，读者如果感兴趣可以自行了解。

# 2.10　以太坊交易

关于以太坊交易，我们一般会将其理解为转账代币，但是其本质上实际是一种广义的交易，交易的内容不仅仅限于转账代币，也可以是转账对象，这个对象就是在使用 Solidity 代码实现智能合约的时候所定义的对象实体。例如，在以太猫应用中，转账的是猫，而不是代币。可以这样理解：交易包含了转账，转账仅是其中的一个可能。交易双方通过地址关联，这个地址就是前面一节中谈到的以太坊的十六进制地址。

本节我们将详细介绍以太坊交易的原理和概念。

## 2.10.1　交易的发起者、类型及发起交易的函数

交易的发起者就是以太坊的使用者，使用者主要有两类：

（1）节点服务，例如 geth 控制台的使用者。
（2）调用节点服务，指 geth 提供 RPC 接口的客户端，例如钱包等。

交易的类型分下面两种：

（1）以太坊 ETH 转账交易。
（2）其他交易，这类交易包含但不限于 ERC20 代币的转账交易。

通常，我们把调用了以太坊节点程序中的“eth_sendTransaction”或“eth_sendRawTransaction”接口所触发的动作或行为称为以太坊交易。目前以太坊 RPC 接口提供了两种标准的交易发起函数，对应上面的交易类型，分为以下两种：

（1）eth_sendTransaction，该函数仅用于以太坊 ETH 转账，参数最终的签名不需要调用者手动进行，它会在当前节点中使用已解锁的发起者 from 的以太坊地址的私钥进行签名，因此每次使用这个函数进行以太坊 ETH 转账时，需要先解锁 from 地址。

（2）eth_sendRawTransaction，需要调用者使用 from 私钥进行签名参数数据的交易函数，目前 ERC20 代币转账交易都是使用这个函数。以太坊 ETH 转账交易一样可以使用“eth_sendRawTransaction”来进行，但转账 ETH 主要由参数控制，这点会在“交易参数的说明”一节中介绍。

为方便阅读，往下的内容中，对于“eth_sendTransaction”和“eth_sendRawTransaction”简称

为“sendTransaction”和“sendRawTransaction”。

## 2.10.2 交易和智能合约的关系

在前面的一节中谈到智能合约中的 transfer 函数，ERC20 代币的转账交易事实上调用的就是智能合约的 transfer 函数，那么智能合约层面的 transfer 函数是如何与节点 RPC 接口层的 sendRawTransaction 联系在一起的呢？本节我们来回答这个问题。

我们知道，ERC20 代币转账交易的第一步是调用 RPC 接口，即调用 sendRawTransaction 接口，在把需要转账的数据传给节点后，节点会提取出每个数据字段，其中就包含 sendRawTransaction 的 data 参数，data 是一个十六进制字符串，它所组成的内容中有部分被称为 methodId，该 ID 对应的就是 transfer 函数的名称转化值，即 transfer 单词通过一定运算后产生的转化值。

有了这个 methodId，等到转账交易被矿工打包处理时就会根据合约地址参数先找到对应的智能合约，合约地址参数由 sendRawTransaction 的 to 参数表示，最后会基于找出的合约去执行数据 data 字段中 methodId 所指示的函数，以及读取这个函数对应的参数数据，例如转账给谁、转多少。

图 2-50 所示是一个转账交易的大致流程图。

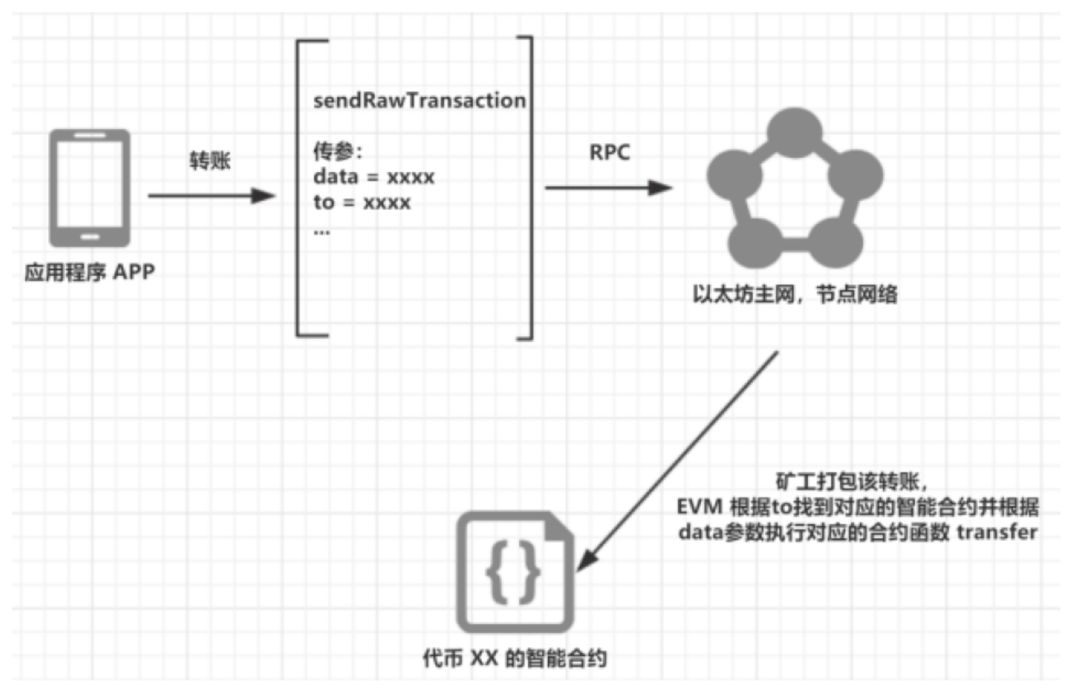



图 2-50 转账交易流程图

也就是说，在应用程序中进行交易并非直接调用智能合约函数，而是先调用以太坊的接口间接调用智能合约的函数。

此外，无论是 sendTransaction 还是 sendRawTransaction，在调用成功后，以太坊都会直接返回一个交易哈希值（全称是“Transaction Hash”，简称 txHash）。注意是直接返回，无须异步等待，但此时还不能确定交易是否成功。

### 2.10.3　交易参数的说明

上一节中我们认识了 sendRawTransaction 中的两个参数，即 data 和 to。除这两个参数之外，在以太坊的交易接口文档中，RPC 接口 sendTransaction 和 sendRawTransaction 的参数还有很多，但其参数的个数是一样的，在这些参数中，地址值类型的参数都是以太坊的合法地址。

图 2-51 所示是以太坊交易接口文档关于 RPC 接口 sendTransaction 和 sendRawTransaction 的参数说明。

- `from` - `String|Number`: The address for the sending account. Uses the web3.eth.defaultAccount property, if not specified. Or an address or index of a local wallet in web3.eth.accounts.wallet.
- `to` - `String`: (optional) The destination address of the message, left undefined for a contract-creation transaction.
- `value` - `Number|String|BN|BigNumber`: (optional) The value transferred for the transaction in wei, also the endowment if it's a contract-creation transaction.
- `gas` - `Number`: (optional, default: To-Be-Determined) The amount of gas to use for the transaction (unused gas is refunded).
- `gasPrice` - `Number|String|BN|BigNumber`: (optional) The price of gas for this transaction in wei, defaults to web3.eth.gasPrice.
- `data` - `String`: (optional) Either a ABI byte string containing the data of the function call on a contract, or in the case of a contract-creation transaction the initialisation code.
- `nonce` - `Number`: (optional) Integer of a nonce. This allows to overwrite your own pending transactions that use the same nonce.

图 2-51　以太坊交易函数的参数

下面我们再来详细认识一下各个参数。

from：代表从哪个地址发起交易，即当前的这笔交易由谁发出。要注意的是，如果交易的 to 是智能合约的地址，那么合约代码中的“msg.sender”变量代表的就是这个 from 地址。

to：代表当前交易的接收地址。注意，这个接收地址不能理解为收款者地址，因为 to 的取值存在下面 3 种情况：

（1）智能合约的地址。

（2）普通以太坊用户的钱包地址。

（3）取空值的时候，代表当前的交易是创建智能合约的交易。

当 to 是第一种情况的时候，当前所发送的交易将会交给对应的智能合约处理，原理和之前谈到的 ERC20 代币转账相同。所以，在进行 ERC20 代币转账时，to 应该是智能合约的地址。

当 to 是第二种情况的时候，就是 ETH 转账，代表把 ETH 以太坊转给哪个地址。

第三种 to 为空的情况，代表当前的交易是部署智能合约到链上的交易。

value：转账的数值。请注意，这个值在使用 sendRawTransaction 进行 ERC20 代币转账时应该是 0。在 ERC20 代币转账时，所要转账的值的多少是定义在 data 参数中的。在使用 sendTransaction 进行 ETH 转账时，value 必须有值，且 value 还是乘上了 $10^{18}$ 次方形式的大数值。

当使用 sendRawTransaction 进行以太坊 ETH 转账交易时，要做到下面 3 点：

（1）to 应该对应收款钱包的以太坊地址。

（2）value 对应的是 ETH 数值，不是 0。

（3）data 参数为空字符串。

只有满足这 3 个条件，sendRawTransaction 进行的交易操作就是以太坊 ETH 转账。

gas：这个 gas 参数就是 gasLimit，但是请不要忘记，在最终交易成功时真实使用的是 GasUsed。交易成功时多出的燃料费会返回，所谓多出的燃料费就是(GasLimit - GasUsed) × GasPrice 部分。

gasPrice：该参数标明每一笔 gas 价值是多少 wei，ETH 与 wei 的换算关系前文已有讲述。所以最终消耗的燃料费应该满足 gas×gasPrice ≥ gasUsed×gasPrice，单位是 wei。

nonce：就是交易序列号。

data：这是一个很重要的参数，既用于交易接口，又用在“eth_call”中。下面以 ERC20 代币转账为例讲解该参数的含义及使用。

首先介绍 data 的格式，data 的格式须满足下面几点：

（1）十六进制格式，例如：

```
0x70a082310000000000000000000000000021af430a036887cb0cfb7083b220f64bb3f8ed8
```

（2）前 10 个字符，包含 0x，是 methodId，它的生成方式比较复杂，是由对应的合约函数的名称经过签名后，再通过 Keccak256 加密取特定数量的字节，然后转为十六进制得出。以下是以太坊版本标准生成 methodId 的代码：

```
    func (method Method) Id() []byte {
  return crypto.Keccak256([]byte(method.Sig()))[:4]
}
```

对于常见的函数其对应的 methodId，有下面的两种：

- 查询余额的 balanceOf 是 0x70a08231。
- 转账 transfer 的是 0xa9059cbb。

（3）前 10 个之后的字符，满足下面的条件：

- 代表的是智能合约中函数的参数。
- 排序方式按照合约函数参数的顺序排列。
- 十六进制的形式。
- 不允许有 0x，即先转成十六进制形式再去掉 0x 字符。
- 去掉 0x 后，每个参数字符个数是 64。

下面举例说明第 3 点：

假设一份智能合约的 transfer 函数的入参是两个整型，其原形是 transfer(uint a,uint b)，那么此时如果要调用这份合约的 transfer 函数，那么 data 的格式应该是：

```
methodId + X + Y
```

其中，X 和 Y 分别对应参数 a 和 b 去掉了 0x 前置字符的十六进制形式，由于 transfer 的 methodId 是 0xa9059cbb，当 a=1、b=2 的时候，data 就是下面的形式：

```
    0xa9059cbb00000000000000000000000000000000000000000000000000000000100000000
000000000000000000000000000000000000000000000002
```

共包括以下 3 部分：

（1）0xa9059cbb

（2）00000000000000000000000000000000000000000000000000000001

（3）00000000000000000000000000000000000000000000000000000002

以上就是以太坊交易函数接口参数的详细说明。请注意，在实际使用时，由于第三方库的封装等原因，可能在使用这些库的时候不需要传这么多参数，但是无论传哪个参数，其含义是不会变的。

下面是以太坊 geth 程序在 ETH 转账时的控制台命令，注意最少要输入 3 个参数：

```
    eth.sendTransaction({from:"0x...",to:"0x...",value:3})
```

### 2.10.4　交易方法的真实含义

在上面一节中，我们已经充分认识了 sendTransaction 和 sendRawTransaction 这两个以太坊发起交易接口的作用和入参。我们要认清的一个事实是，上面的两个交易接口所指的“交易”代表的不仅仅是代币的转移，还代表以交易的形式访问智能合约的一个公有函数，被访问函数所产生的变化会被记录到区块数据内，而控制访问函数的方式是通过入参 data 来实现的。

这是什么意思呢？意思就是，在调用 sendRawTransaction 发起以太坊交易时，如果所传参数中的 data 是 methodId，而不是 transfer，将不会实现代币转移，即无法达到代币转账的目的，而其达成的效果最终由智能合约函数所定义的代码来决定。

为什么是 sendRawTransaction 而不是 sendTransaction？因为 sendTransaction 已经被以太坊源码封装好了，它只能用来转账以太坊 ETH，本质上和 sendRawTransaction 是一样的，被封装好了的 sendTransaction，此时它的 data 被设置成了发起交易的附属信息，类似于备注。

下面再通过一个例子来阐述上述内容。假设智能合约 A 中定义了一个函数，其名称是 setName，这个函数的功能是设置名称，假设此时 setName 的 methodId 是 0xabc，入参是一个字符串。那么当我们使用 sendRawTransaction 调用智能合约 A 中的 setName 函数时，data 参数就要设置为与 setName 相关的数据，待发起交易时，这笔交易被矿工成功打包进区块中之后，名称便成功地被设置了，且结果也会被记录到这笔交易所打包进了的区块中——被持久化到节点中的“键-值对”<k,v>数据库中。此时，这笔交易只是调用了合约中的 setName 函数，仅达到了一个设置名称的目的，并没有发生任何的代币转账，但我们也把这一次调用看作是一次交易。

下面是我们在实际开发中使用 sendRawTransaction 进行交易时经常用到的调用方法的名称：

- 转账，此时 data 中转化后的 methodId 原型对应的是 transfer。
- 授权，此时 data 中转化后的 methodId 原型对应的是 approve。

## 2.10.5 交易的状态

当我们使用 sendTransaction 或 sendRawTransaction 将一笔交易提交到以太坊，并得到了以太坊返回的哈希值后，这时我们并不能判断这笔交易的最终结果是成功还是失败。请记住，获取了哈希值只能代表以太坊成功地接收了这笔交易的请求，不能代表交易最终是否成功。

在交易被以太坊成功接收后，它会经历图 2-52 所示的生命周期。

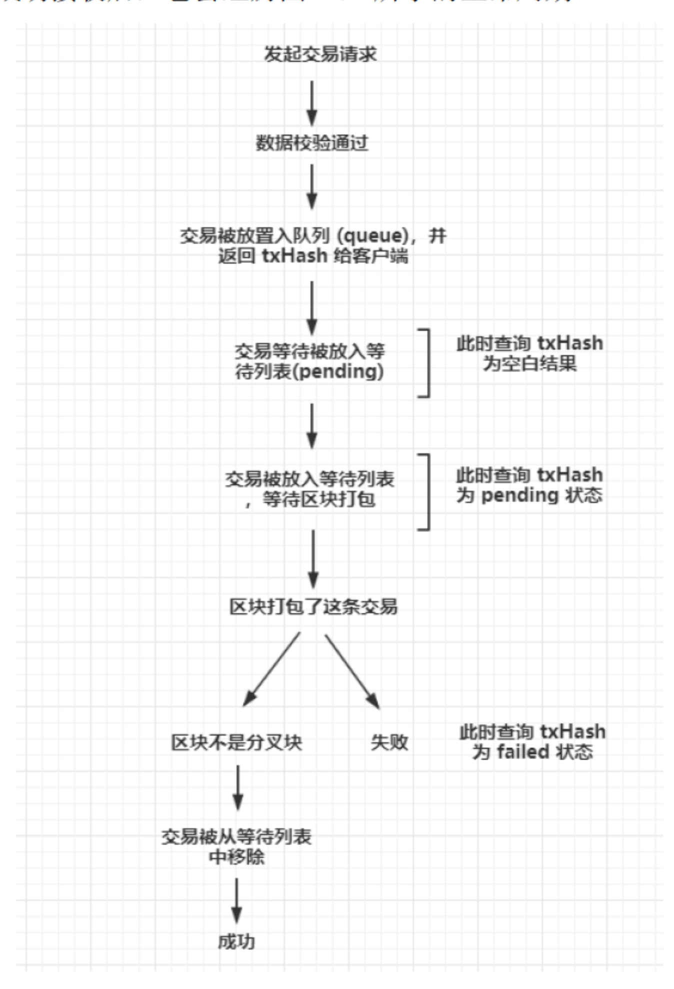



图 2-52　一次交易从发送到彻底成功的生命周期

图 2-52 中的 txHash 就是交易的哈希值，可以看出，一次交易在成功提交到以太坊后共有 4 种状态，分别是：

- Unknown（未知状态）。还没被放入到 txPool 以太坊交易池中，这个时候如果用区块链浏览器查询这个 txHash，就会发现无任何信息。
- Pending（等待或挂起状态）。这个状态是最常见的，是交易成功的必经状态，此时我们用区块链浏览器查询，能查询出部分交易信息。注意是部分交易信息，例如图 2-53 所示的查询结果并没有显示区块号信息，即 “block height” （区块高度）。




图 2-53　用区块链浏览器查询 Pending（等待或挂起）状态下的交易时所看到的信息

- Success（成功状态）。代表交易成功。
- Failed（失败状态）。注意，在交易失败时，也能够查询出该交易的相关信息，例如区块高度等，如图 2-54 所示。

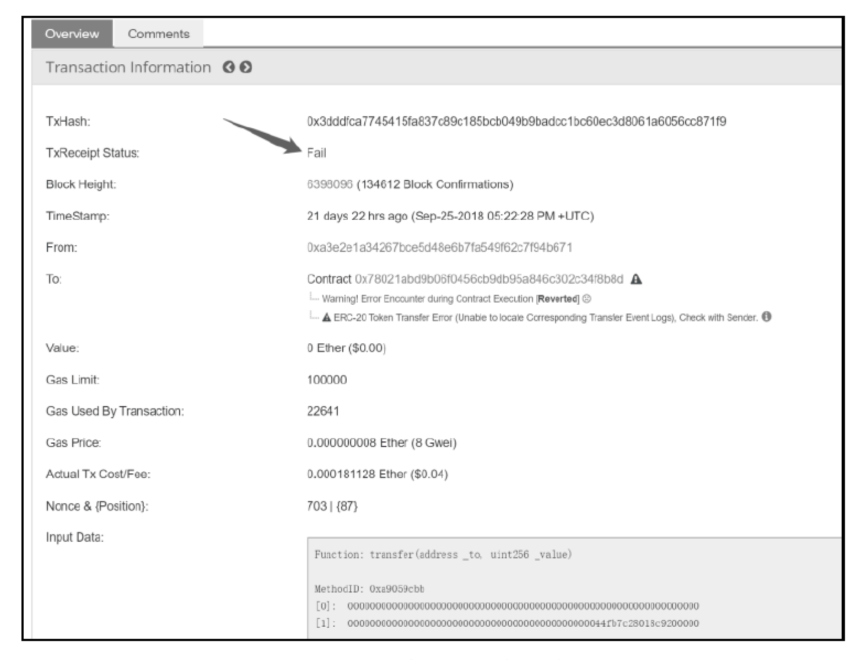



图 2-54　失败状态的交易信息

因为以太坊交易池的大小是有限制的，所以常常会造成一些交易订单只是处于被放置到交易池，尚未被交易的状态，该状态称为“Unknown”（未知状态）。造成这种情况的原因是，矿工在交易池中的交易订单的排序算法受 GasPrice 的影响，也就是说，如果交易订单 A 此刻排在第三，刚好有新的订单 B 进来了，且 B 的 GasPrice 很高，那么订单 A 就有可能被排后。根据这个特点，如果长时间地出现这种排队的情况，就有可能导致某个低 GasPrice 的订单一直处于 Pending（等待或挂起）状态，迟迟不被矿工打包，从而会出现有些等待状态的交易订单被“挂起”几天甚至更久时间的情况。

Fail 的失败情况一般发生在和智能合约交互的相关交易中，交易的错误由合约的代码抛出，比如参数错误等原因。

## 2.10.6 交易被打包

上面我们认识了交易从发送到添加进以太坊交易订单池的过程。那么被添加到了交易池中的交易最终又是怎样被打包进区块里面的呢？下面我们从如图 2-55 所示的流程图来认识一下。

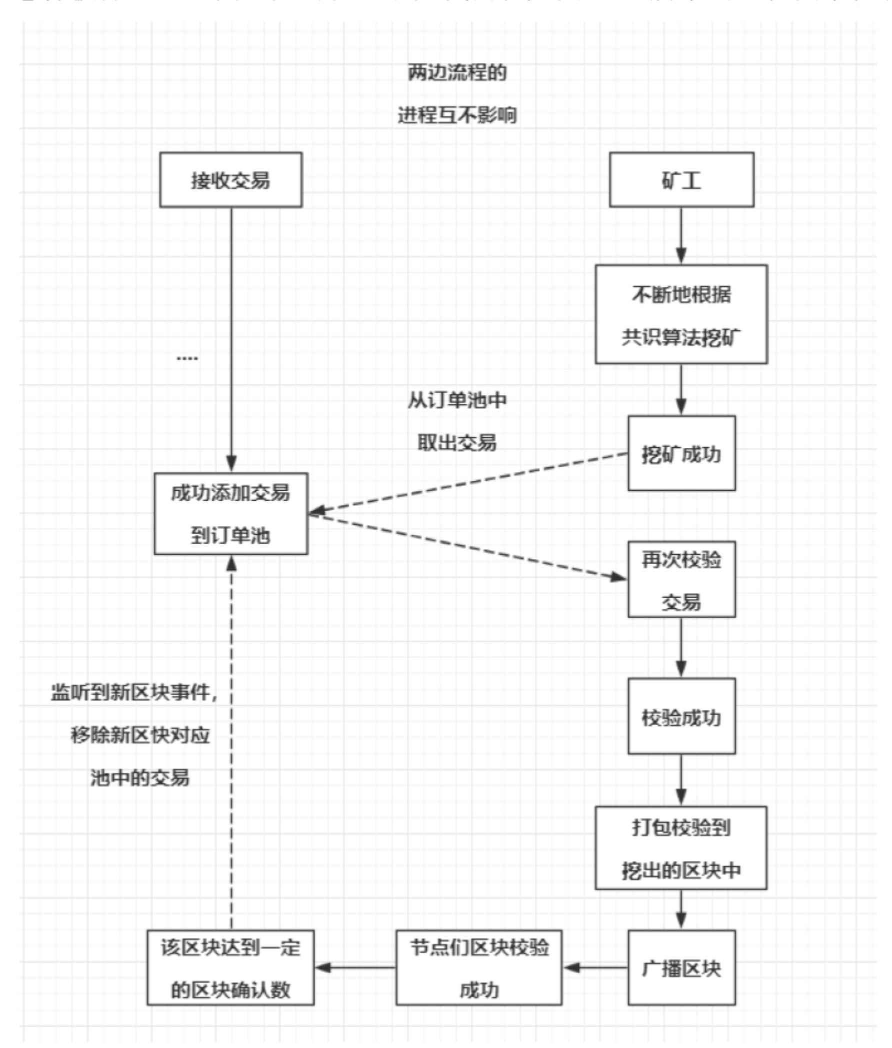



图 2-55　交易从发送到添加进以太坊交易订单池的过程

图 2-55 便是以太坊订单池中的交易订单被添加到池中之后，再被打包到区块中直至被从订单池移除的一个大致的生命流程图。

其中对于矿工打包交易时“再次校验交易”的步骤，内部拥有一次交易燃料费的计算步骤，这是在前面“燃料费”一节中，EVM（虚拟机）计算燃料费的流程。

# 2.11　“代币”余额

和以太坊交易一样，以太坊的“代币”（Token）余额并非仅仅局限于代币，例如，以太坊的“以太猫”应用，我们基于它的智能合约查询余额时，得到的结果代表的就是“该地址拥有多少只猫”。为了便于文字表述，我们还是称 Token 为代币。

以太坊的“代币”余额，主要有下面的两种类型：

（1）ETH 余额。

（2）智能合约代码定义的对象的拥有数。

不同的“代币”类型，其查询余额的方式也不同。以太坊“代币”余额一般是通过某个以太坊地址来查询的，主要有以下 3 种查询方式：

（1）通过调用以太坊的接口查询。

（2）使用以太坊浏览器进行查询。

（3）使用以太坊钱包 App 查询。

第二、三种方式的本质也是通过调用以太坊接口进行查询，不同之处在于这两种查询帮我们封装好了代码层面的东西，查询操作可直接在应用层进行。

查询余额的接口也分为两类，分别是：

（1）以太坊的 ETH 余额查询接口“eth_getBalance”。

（2）以太坊的“eth_call”接口。使用“eth_call”访问智能合约提供的余额查询函数来达到查询的目的，例如 ERC20 标准提供了“balanceOf”函数。

这两类接口有很大的区别。针对 ETH 查询，以太坊提供了一个专门的接口“eth_getBalance”，就像 sendTransaction 和 sendRawTransaction 一样，都提供了一个专门的接口。而其他的非 ETH 的“代币”余额查询，包含 ERC20 代币和非代币资产，只能调用以太坊提供的一个万能接口“eth_call”来查询，且在查询时必须传入正确的 data 参数。

如图 2-56 所示是 ERC20 代币的查询请求示例。

请留意图 2-57 中的 data，它的前 10 个字符就是我们在“以太坊交易”一节中讲到的“balanceOf”的“methodId”，后面跟随的参数就是我们所要查询余额的以太坊地址。

图 2-57 的“method”键对应的值是“eth_call”，它的详细介绍可参考“重要接口的含义详解”一节，目前智能合约的非转账类函数都能通过这个接口进行调用，只需要把合约中对应函数的“methodId”标明正确和入参设置到 data 中即可。

为什么是合约中的非转账类函数呢？如果我们在实际的开发中使用 eth_call 来调用合约中的 transfer 转账函数会怎样呢？

答案是以太坊的 eth_call 用来调用智能合约的 transfer 函数，既不报错也不会实现真正的数值转账，最终返回的结果是一个“0x 字符串”。




图 2-56　使用 Postman 工具查询代币的余额

# 2.12　以太坊浏览器

以太坊浏览器是区块链浏览器中的一类。可以这样说，区块链是一个大的概念，区块链浏览器包含比特币浏览器和以太坊浏览器等。

我们之前介绍的余额查询方式是面向开发人员的，对于普通用户来说，要进行以太坊相关信息（包含代币余额）的查询，一般使用的都是以太坊浏览器。以太坊浏览器其实就是一个网页应用程序，也就是对网站访问，其内部的查询功能是通过封装好了的以太坊接口实现的。目前最为权威并出名的以太坊浏览器是“etherscan.io”，官方网站是 https://etherscan.io/ 。

“etherscan.io”几乎具备了以太坊链上所有数据信息的查询功能，同时还提供了最新区块生成记录及其最新交易的信息列表，如图 2-57 所示。

在上述区块链浏览器的主页中，右上角的输入框支持以下各项查询功能：

- 钱包地址的查询，需要输入要查询的以太坊地址（Address）。
- 交易详情的查询，需要输入要查询交易的 txHash。
- 区块信息的查询，需要输入要查询的区块的哈希值。
- 代币的信息查询，需要输入要查询代币的地址，它也是一个以太坊地址。




图 2-57　Etherscan 浏览器主页的最新区块生成记录

如图 2-58 所示是地址为“0x78021abd9b06f0456cb9db95a846c302c34f8b8d”的代币查询结果图。

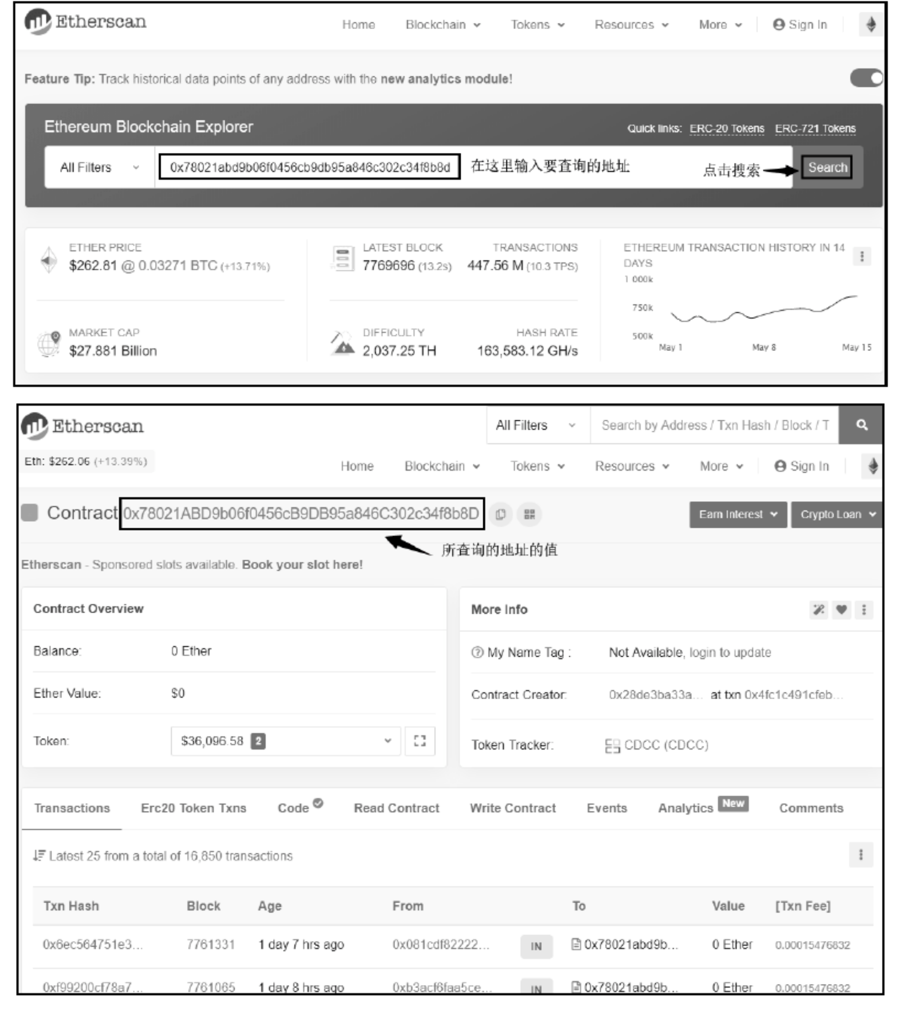



图 2-58　区块链浏览器查询某个以太坊地址显示的页面

此外，还有我们在开发过程中经常会进行交易哈希值的查询，该查询显示的页面和在“交易的状态”一节中的图 2.53 和图 2.54 一样。一般来说，查询交易哈希值的目的主要是为了了解交易的状态，比如观察交易是等待（Pending）被打包还是已经成功了（Success）等。

## 2.12.1　区块链浏览器访问合约函数

上面一节提到可以在以太坊浏览器中查询代币余额，这其实只是调用合约代码函数的一种方式。如果我们要查询的是以太坊，那么直接在浏览器的输入框中输入要查询的钱包地址即可。例如，在图 2-59 中，左上角的“Balance”字段对应的就是当前被查询地址中以太坊 ETH 的数值（即余额），单位是 ETH。

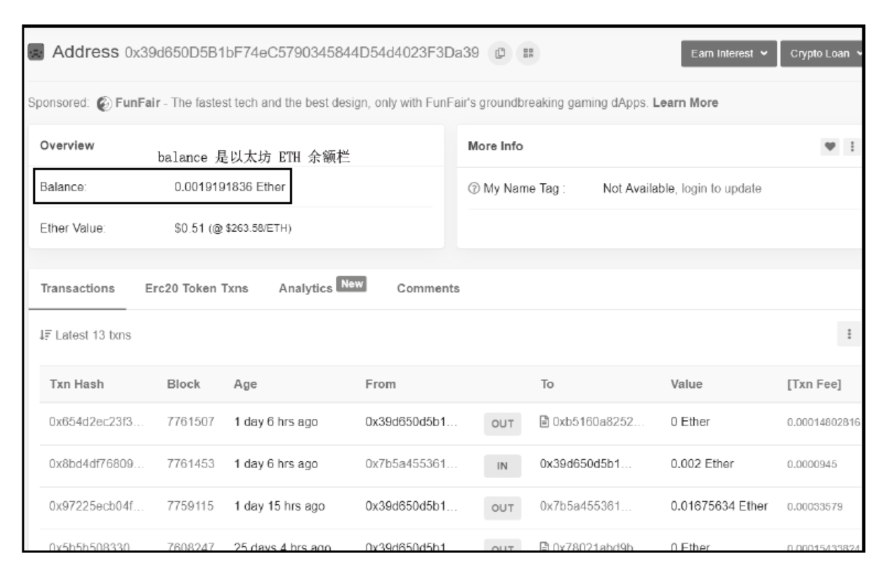



图 2-59　区块链浏览器查看以太坊地址的 ETH 余额

请注意，上面是查询 ETH 余额的方式，如果要查询的不是 ETH 的余额，而是某一种 ERC20 代币的余额，那么就需要进入对应的 ERC20 代币界面去查询，这是什么意思呢？例如，要查询某个钱包地址所拥有 ERC20 代币 CDCC 的个数是多少，首先要找到这个 ERC20 代币的合约地址，从以太坊浏览器中查询该地址，进入到该代币的详情页面后才能进行下一步的查询，这里的 CDCC 是这个代币的 Symbol，意思是符号。下面给出查询步骤：

（1）找出要查询的 ERC20 代币的合约地址，例如：

```
0x78021abd9b06f0456cb9db95a846c302c34f8b8d
```

（2）从以太坊浏览器中查询上面的地址，进入到该代币详情页，如图 2-60 所示。

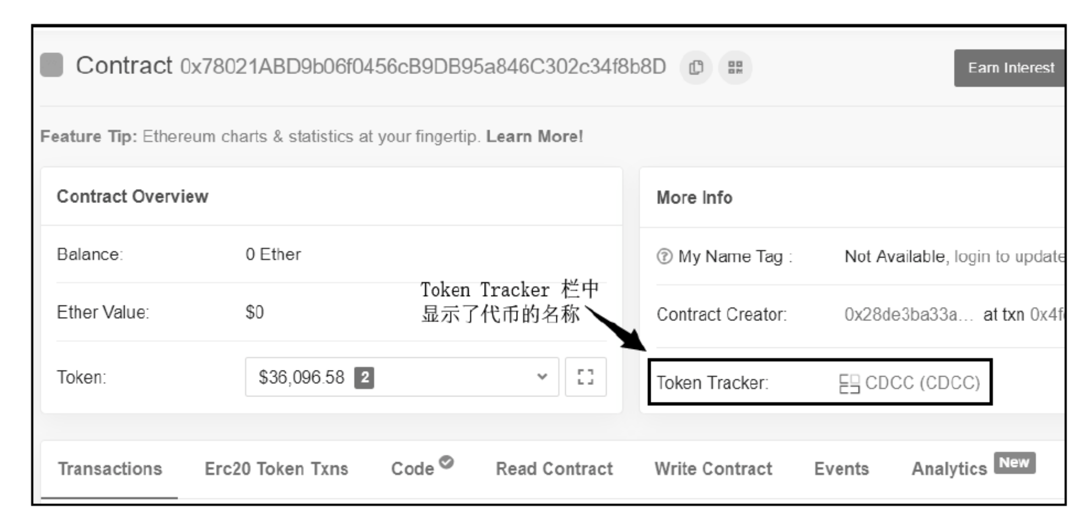



图 2-60　用区块链浏览器查看 ERC20 代币

（3）单击界面上的“Read Contract”按钮，从显示出的内容中可以直观地看到之前在 ERC20 标准一节中所认识的一些智能合约中的字段变量，例如 name、totalSupply 等，如图 2-61 所示。

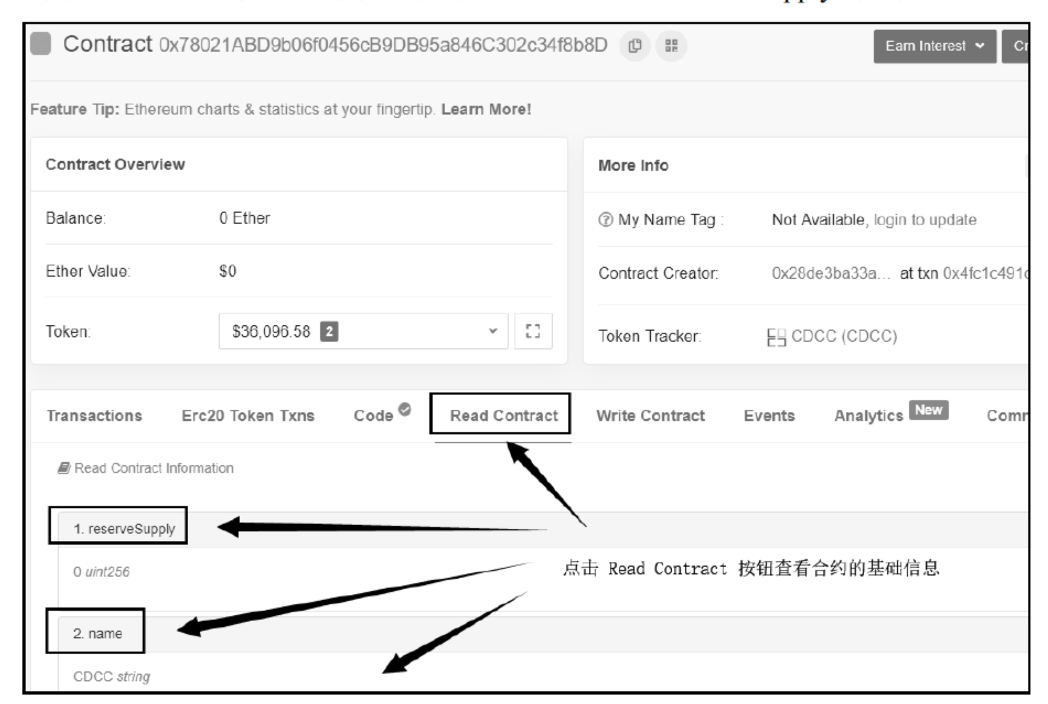



图 2-61　用区块链浏览器查看 ERC2 合约标准的字段

（4）在“Read Contract”页面显示区域中，向下滑动鼠标，直到看见 ERC20 标准中的“balanceOf”代币余额查询函数，如图 2-62 所示。




图 2-62 在区块链浏览器中调用 ERC20 标准合约的余额函数

（5）在“balanceOf”函数下面的输入框中输入要查询的钱包地址，然后单击“Query”按钮，就能对此钱包地址拥有多少个 CDCC 代币余额进行查询，如图 2-63 所示。




图 2-63 查询 CDCC 代币余额

在上面最终查询出的余额值是一个乘上了当前代币的 $10^{decimal}$ 次方数值的值。如果要得出实际拥有的以“个”为单位的代币值，记得要将这个大数值除以 $10^{decimal}$，此外这个 decimal 的值也是能够在“Read Contract”页面看到的。如图 2-64 所示。




图 2-64 查看 decimal 的值

上面通过在以太坊浏览器进行代币余额查询的一个例子，来说明了如何在以太坊浏览器中“调用合约函数”。除了余额的查询外，还可以查询授权值等，它们的操作方式大同小异，如图 2-65 所示。

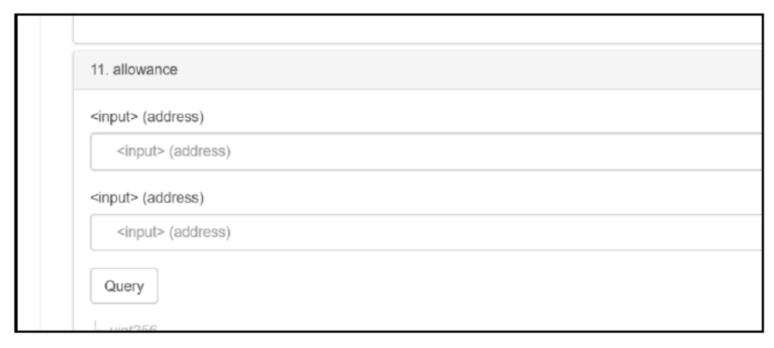



图 2-65　在区块链浏览器调用 ERC20 标准合约的授权函数

## 2.12.2　区块链浏览器查看交易记录

在区块链浏览器中查看交易记录也是一项基本技能，仍然以“etherscan.io”为例进行演示。在“etherscan.io”主页的搜索输入框中，输入想要查询的以太坊地址进行搜索，就能进入到对应的地址主页面，如图 2-66 所示。

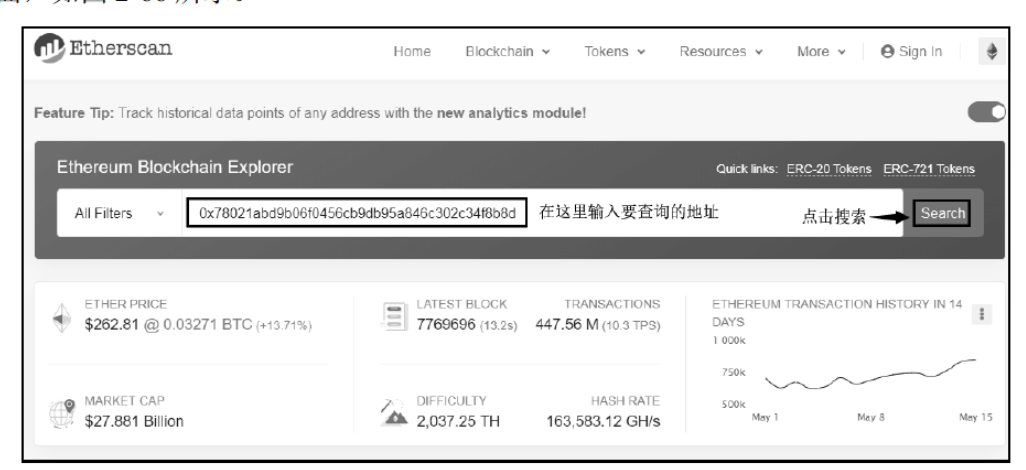



图 2-66　在“etherscan.io”主页输入要查询的以太坊地址

例如，要查询地址 0xB0bF3242eBED6525d256B5B32BD69C7EAc63F6F2 的交易记录，在进行搜索后可以看到交易记录的列表页面。如图 2-67 所示，可以选择查看具备多类交易的交易信息项。

- 类型“Transactions”对应的是以太坊 ETH 代币交易的交易记录。
- 类型“Erc20 Token Txns”代表的是“ERC20”代币的交易项，鼠标单击即可进行切换。

此外，如果还有其他类型的“代币”，例如“ERC721”类型的交易记录，那么它将会出现在“Erc20 Token Txns”按钮的右边，以此类推。在列表中，每一条蓝色字体都可以用鼠标单击，我们单击“TxHash”列下的每笔交易哈希值，就能进入到被单击交易所对应的详情页。

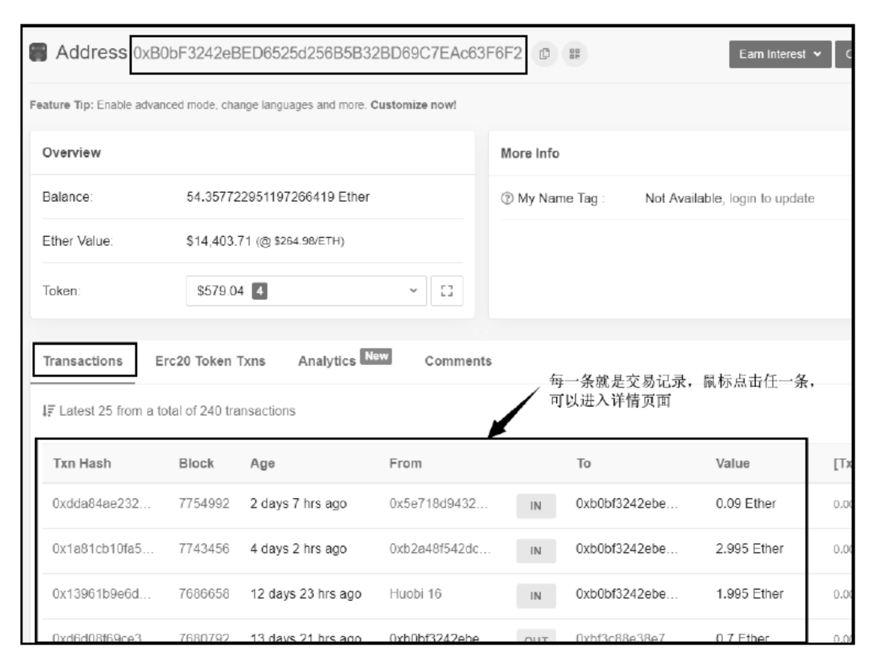


图 2-67　交易的详情页

## 2.12.3　非 ETH 交易记录不能作为资产转账成功的依据

对于非 ETH 交易来说，一般大家都会以为如果一笔交易记录在区块链浏览器上能被查询到，且状态是成功的，就认为这笔交易背后所转移的资产也就到账了。

事实上，这种判断是错误的。请记住，区块链浏览器的非 ETH 交易记录不能作为资产转账成功的依据！

为什么呢？

依然以“etherscan.io”为例，在上一节中，我们可以根据一个以太坊地址在“etherscan.io”上查询到与它相关的交易记录。那么“etherscan.io”网站是怎样从以太坊区块链上获取到这些交易的呢？答案是，“etherscan.io”上的非 ETH 代币的交易记录都是通过读取区块“Event log”（事件日志）数据中的“Event”得到的，而“Event”是由我们编写的智能合约的代码生成的。

在 ETH 的交易记录中并没有“Event log”这个概念，所有在正规区块链浏览器中看到的 ETH 交易记录都可以作为 ETH 交易的依据。

下面分别指出“Event”相关的几个技术点：

- “Event”是开发者可以在智能合约代码中随意定义并在函数中触发的事件。
- 我们可以在智能合约代码中定义“Transfer”的“Event”。
- 只要被记录到了区块的“Event log”中的“Event”，就能被区块遍历器读取。
- 区块中的“Transfer Event”会被“etherscan.io”当作交易记录读取并显示。
- 所有经过以太坊“sendRawTransaction”接口调用触发的智能合约函数，其内部的“Event”将会被记录到区块的“Event log”中，从而被区块遍历器读取。下面举个例子阐述这个结论。

智能合约 A 拥有函数“func1”，它仅仅触发了一个“Transfer”的“Event”，而没有做其他的操作。

```
    function func1(address _from,address _to, uint256 _value) public returns (bool success) {
        emit Transfer(msg.sender, _to, _value);
        return true;
    }
```

如果此时我们使用“sendRawTransaction”接口调用了智能合约 A 的这个函数，就会造成“etherscan.io”把里面被触发的“Transfer Event”从区块中读取出，然后当作一笔交易记录显示在“msg.sender”地址所对应的交易记录页面中，“_from”参数将会对应显示到网页页面的“from”标签中，而“_to”就是收款人的地址标签。

但是，请注意，我们在“Func1”函数中并没有添加任何的资产转移代码，例如添加了下面的合法代码：

```
    balances[_to] += _value;
```

上面例子最终导致的结果是，在浏览器中能查看到交易的成功记录，但是却没有引发真实的“代币”资产转账。

如果我们不采用“sendRawTransaction”接口调用智能合约 A 的这个函数，而使用“eth_call”来调用，它里面的“Event”将不会被记录到“Event log”中。

那么如何判断一笔交易记录是有效的呢？判断一笔交易记录是否有效的必要条件是：

- 当前交易对应的“Event”是可以被查询到的。
- 到对应的智能合约中查看触发“Event”的函数代码。
- 保证触发“Event”的合约函数代码是没有问题的，有明确的资产转移代码，例如 balances[_to] += _value 。

上述判断条件是从一个广义的角度来进行的，还可以进一步细分来添加其他的判断条件，例如保证合约遵循了 ERC 系列标准的判断条件，这样我们就能根据标准的函数名称查询对应的值，例如“ERC20”标准中查询余额的函数，其名称为“balanceOf”。

细分的条件都不是必要的，以合约的标准为例，它本身只是一种规范，我们可以不遵循这种规范实现自己的“代币”合约，查询余额的函数名称也不一定就是“balanceOf”，也可以定义为“getBalance”，只要能正确地实现余额查询功能即可。

回到我们上面的示例代码中，例子中的函数触发了“Transfer Event”（转账事件）却没有引起真实资产转账的合约代码，我们一般都认定为是有代码问题的智能合约。而目前在绝大部分流通的“代币”中，智能合约几乎都满足凡是触发了“Transfer Event”的函数，其内部必然是进行了资产转移。

还有，对于在虚拟资产交易所中可以进行交易的“ERC20”代币来说，它们的智能合约代码都已被交易所检查过，也就不存在我们上面所举例的情况。

## 2.12.4　区块链浏览器查看智能合约的代码

在上一节中，我们介绍了如何判断一笔交易记录的有效性——我们只需要到智能合约中查看合约是否触发了“Transfer event”函数代码即可得出结论，在区块链浏览器中，同样也可以查看合

约代码。

依然以“etherscan.io”为例，首先假定已得知对应智能合约的以太坊地址，进行搜索，假设“CDCC”代币的地址是：

```
0x78021abd9b06f0456cb9db95a846c302c34f8b8d
```

如图 2-68 所示是查询的主页面。单击“code”按钮，就能看到当前合约的所有代码，如 2-69 所示。

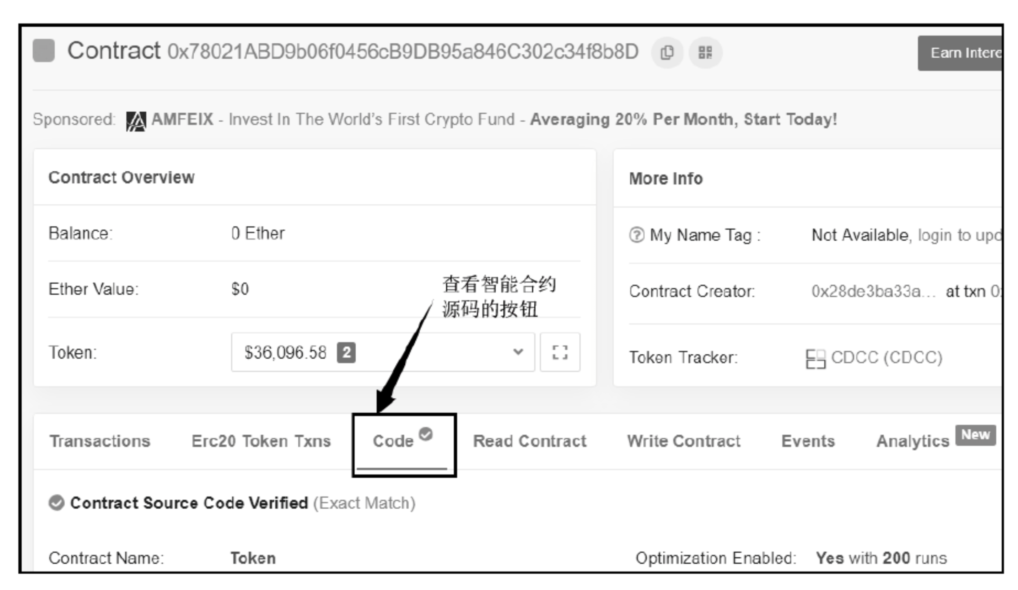



图 2-68　查看合约的代码




图 2-69　合约的代码内容

当我们在交易详细页面查看“Event log”，获取触发“Event”的函数名称后，就能够到合约代码页面查看该函数的代码，如图 2-70 所示。




图 2-70　查看某笔交易的事件日志（Event Log）

# 2.13 以太坊零地址

在以太坊区块链中存在一个零地址，即它的十六进制值为零，原形是：

```
0x0000000000000000000000000000000000000000
```

这个地址拥有下面的特点：

- 与创世区块无关。
- 当启动以太坊挖矿程序，也即在节点代码中启动“矿工”挖矿时，如果没有设置挖矿的收益地址，就会默认使用零地址挖矿，挖矿所得的 ETH 将归零地址所有。
- 是一个合法的以太坊地址，能够接收代币的转账。
- 它的私钥可能仍然没被碰撞出。

在区块链浏览器中查询这个地址，观察它的以太坊 ETH 交易记录，可以发现，其所有的 ETH 交易记录都只有转入而没有转出的记录。据此可以猜测，它的私钥还没有被碰撞出，为什么这样说呢？

因为每个合法的以太坊地址都对应有一个私钥，只要满足地址格式，我们不需要知道它的私钥是什么就可以使用，可以向它交易转账、查询它的各种操作记录等。

零地址便是如此，因为它很容易被记住及写出，全部数值为 0 即可，不需要刻意地去进行运算得出，但它亦始终对应有一个私钥。在私钥生成公钥的算法中，我们知道如果要根据一个公钥来逆推出私钥，概率非常小，但不是不可能，只要概率不为 0，就不能说不可能。

加上它所有的 ETH 交易记录都是只有被转入而没有转出的记录，最难以想象的情况就是零地址的私钥已经被碰撞出了，但是拥有者还不打算转出任何一个 ETH。

综上所述，零地址的私钥可能还没被碰撞出。

## 2.13.1 零地址的交易转出假象

上面一小节讲到，在零地址的私钥还没被碰撞出的时候是不可能有人使用零地址去做交易转出操作的。但是，当我们在区块链浏览器中查看零地址的非 ETH 交易记录时却能够看到存在“Out”（转出）的情况，如图 2-71 所示。

这种情况怎么解释呢？这是零地址交易转出的假象。回顾“浏览器的交易记录不能作为非 ETH 的交易依据”一节中所讲到的内容就能明白。

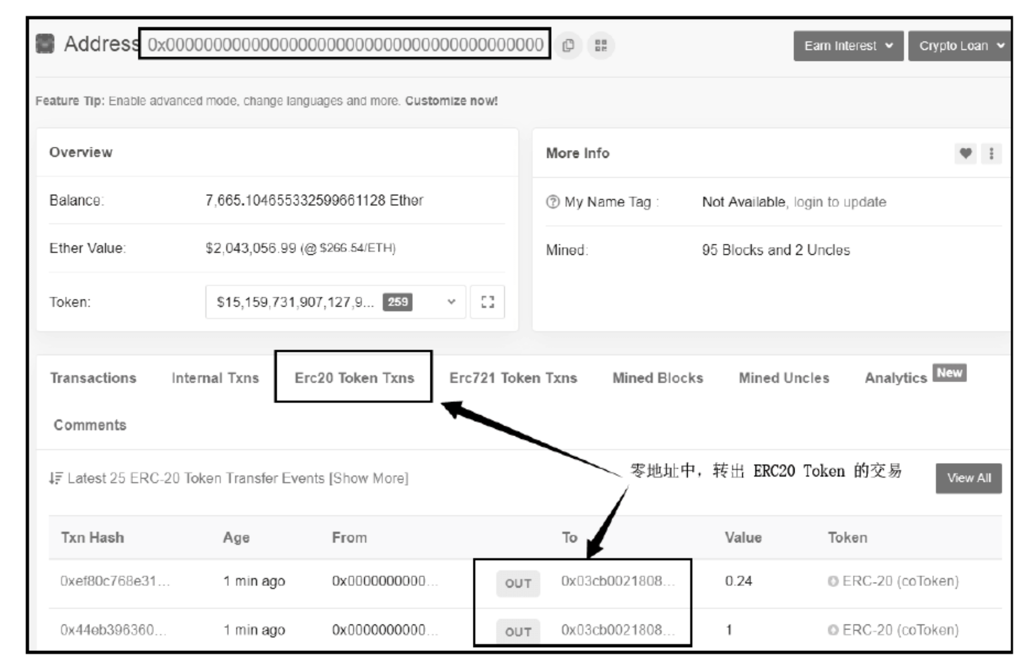



图 2-71 零地址的非 ETH 交易记录存在“Out”的情况

其原因是，零地址在智能合约的触发“Transfer event”事件的代码中被开发者设置为了“Event”的“from”参数。例如，下面的代码中，“address(0)”就被当作了“Transfer event”的“from”参数，当函数“mint”被“sendRawTransaction”调用后，所触发的“Event”就会被记录到区块中，最终被区块链浏览器获取到而显示在交易记录列表页面中，页面所显示的“from”就是“mint”函数中“Transfer”事件的入参“from”值。

```
    function mint(address _to, uint256 _amount) onlyOwner canMint public returns
(bool) {
        totalSupply_ = totalSupply_.add(_amount);
        balances[_to] = balances[_to].add(_amount);
        Mint(_to, _amount);
        Transfer(address(0), _to, _amount);  // 这一行触发事件
        return true;
    }
```

上面的合约函数达到了一个给指定的以太坊地址添加资产的目的，“_to”是接收者，“_amount”是数量。

在上面的流程中需要注意的是，负责使用私钥签名以太坊“sendRawTransaction”接口函数的不是零地址的，而是谁调用这个接口函数谁就签名。这说明在区块链浏览器的交易详情页面中显示的“from”数据项和进行签名的地址是没有关系的，仅和“Transfer Event”的“from”入参值有关，即“Transfer Event”的 from 才是区块链浏览器交易详情页面中显示的“from”。

那么零地址是否可以转出资产呢？答案是可以的。对于签名不使用零地址的交易，只要满足从零地址的余额中减少资产，然后将减少的资产累加到收款地址即可。例如下面的函数，只要它被“sendRawTransaction”指定调用后就能够实现从零地址转出资产，前提是我们得在其他地方先给零地址赋予资产。

```
    constructor() public {
```

```
        balances[0x0] = 1000000000;  //给零地址赋予代币资产
    }
    function release(address _to,uint256 _value)public returns (bool success){
        require(balances[0x0] >= _value);
        balances[0x0] -= _value;
        balances[_to] += _value;
        emit Transfer(0x0, _to, _value);
        return true;
    }
```

## 2.13.2 零地址的意义

以下是零地址使用最多的两种场景：

（1）用于启动以太坊挖矿程序，如果没有设置挖矿的收益地址就默认使用零地址挖矿。

（2）在智能合约的代码编写中使用零地址，例如作为函数的参数。

在第一种场景中，零地址充当的是一个默认统一处理的方式。比如，挖矿过程中没有设置收益地址，这个时候应该怎么办呢？一种做法是强制不允许用户进行挖矿操作，要求必须设置收益地址。另一种做法是不强制，可以继续挖矿。此时就需要一个默认的收益地址，那么这个地址设置为谁合适？毫无疑问，选择零地址是最为合理的，就表现形式上来看，零代表的就是开始，也很容易被记住。

在第二种场景中，如果我们要给某个地址直接赠送“代币”资产或者是表现为生成“代币”资产，在操作完之后，必须触发转账事件“Transfer event”，这就需要一个“from”参数来表示从哪儿转账出去的。但是，基于从无到有再转移的过程，并不存在从某个确切的拥有资产的地址转账到另一个地址的过程，这个“from”选择为零地址最为合理。

回顾前面合约函数的例子：

```
    function mint(address _to, uint256 _amount) onlyOwner canMint public returns
(bool) {
        totalSupply_ = totalSupply_.add(_amount);
        balances[_to] = balances[_to].add(_amount);
        Mint(_to, _amount);
        Transfer(address(0), _to, _amount);
        return true;
    }
```

该例就实现了往一个给指定的以太坊地址添加资产的目的，添加的形式是直接添加，不存在从一个地址转账给收款地址。这个时候触发转账事件“Transfer event”，“from”参数选择的就是零地址。

因此，我们可以将零地址看作是某些情况下的合理默认值，就好比我们在编程时，初始化一个“int”类型的变量，其默认值总是 0 一样。

# 2.14　小 结

第 2 章可以说是全书最为重要的一章，囊括了以太坊基础性的知识点，整体介绍了以太坊的技术模块，详细地讲解了区块的组成及其内部各个字段变量的作用，对以后实现以太坊相关接口起到先行作用。

同时也从比特币的 UTXO 模型引申出以太坊的账户模型，介绍了以太坊的 MPT 树与默克尔树和字典树的特性，以及 MPT 在账户体系中的应用。

特别地，本章还讲解了以太坊的“Ghost 协议”，以及以太坊公链在分叉网络中，根据“Ghost 协议”选择最优链，而非比特币的最长链规则。由该协议所衍生的叔块知识也有对应的讲解，包含叔块的定义、打包规则和奖励规则。

本章也先行简介了以太坊代表性的智能合约模块，并列举了两个经典的合约标准——ERC20 与 ERC721。在第 3 章我们将会进一步对合约模块进行学习。

在以太坊交易模块一节中，“交易参数的说明”与“交易方法的真实含义”这两节内容尤其重要，通过这两节的学习，我们可以深刻地理解以太坊交易的两个核心接口“sendTransaction”和“sendRawTransaction”的区别及其交易的实际含义，交易并不一定就是代币的转账，它还能做其他事情。

另外，关于以太坊浏览器的基本使用，也在本章做了相应的介绍。

最后，还介绍了“非 ETH 交易记录不能作为资产转账成功的依据”和“零地址的交易转出假象”两个较冷门的知识点，以扩大读者的知识面。

# 第 3 章

# 智能合约的编写、发布和调用

智能合约的概念不仅仅以太坊具备，其他的公链也同样具备智能合约机制，例如 EOS 公链等，它是区块链技术的一个模块。

在第 2 章中，我们介绍了智能合约的部分知识点，本章我们将从一个整体的角度来进一步认识智能合约。

## 3.1 智能合约与以太坊 DApp

以太坊的智能合约功能模块可以让开发者自由地使用特定的计算机语言来编写智能合约，并以代码文件的形式呈现，如果在以太坊网络上发布这份智能合约，此时的智能合约便变成了一个智能合约程序。

这个程序运行在以太坊上，拥有自己独特的功能，这些功能也能让别人使用。例如，以太猫游戏、以太僵尸游戏等，它们的原形都是发布在以太坊上的一份智能合约，这份智能合约发布之后，就会被广播到整个节点网络的节点中，形成分布式应用。

这些基于智能合约的以太坊应用，还有另外一个名称，就是“以太坊智能合约 DApp”，全称是“以太坊智能合约分布式应用”。

以太坊的 DApp 有很多种类型，有基于智能合约实现的应用，也有基于以太坊功能接口实现的应用，比如钱包类等，统称为以太坊 DApp。

所以，智能合约仅是以太坊 DApp 实现的方式之一。以太坊 DApp 的组成如图 3-1 所示。

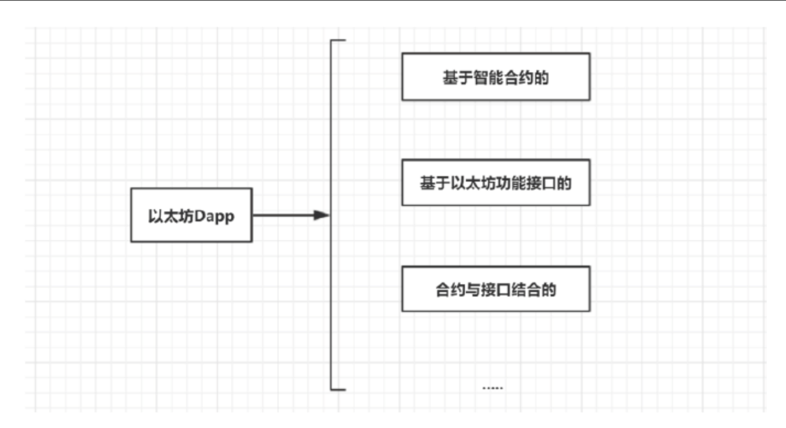



图 3-1　以太坊 DApp 的分类

# 3.2　认识 Remix

智能合约的编写需要使用计算机语言，目前常用的是 Solidity 语言。Solidity 也是 Ethereum 官方设计和支持的编程语言，专门用于编写智能合约。关于 Solidity 编程语言的知识，读者可以参看相关资料。本节我们主要从智能合约编写工具 Remix 的基础使用以及智能合约的编写方法这两方面进行讲解。

编写智能合约的工具有很多种，笔者推荐使用以太坊官方推出的 Remix。

Remix 是一个开源的 Solidity 智能合约开发环境，它提供了基本的编译、部署至本地、合约测试和执行合约等功能，它的开源地址是 https://github.com/ethereum/remix。我们可以从这个地址中把 Remix 的源码下载下来，然后自行在本地编译运行，但是这个过程可能会遇到很多编译上的问题，例如环境配置问题等。建议大家使用 Remix 的网上浏览器版本，这样就不用自己下载源码到本地再进行编译了，可以直接通过浏览器打开链接进行智能合约的编写，Remix 网上浏览器的链接为 https://remix.ethereum.org/。

在浏览器中打开上面的链接，就会看到如图 3-2 所示的界面，这就是 Remix 的主页。

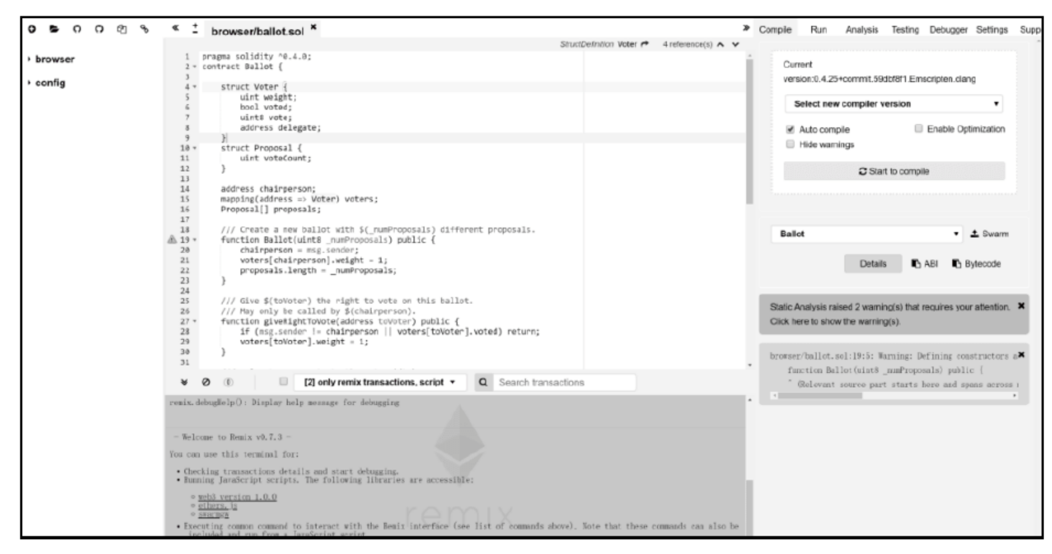



图 3-2　Remix 的主页，就是它的主界面

在主页面左上角的图标中，我们要知道其中 3 个选项所代表的含义："+"图标代表创建一个新文件，形如文件夹的按钮表示从电脑打开一个文件，"browser"是默认的存放所有新建文件的文件夹，如图 3-3 所示。

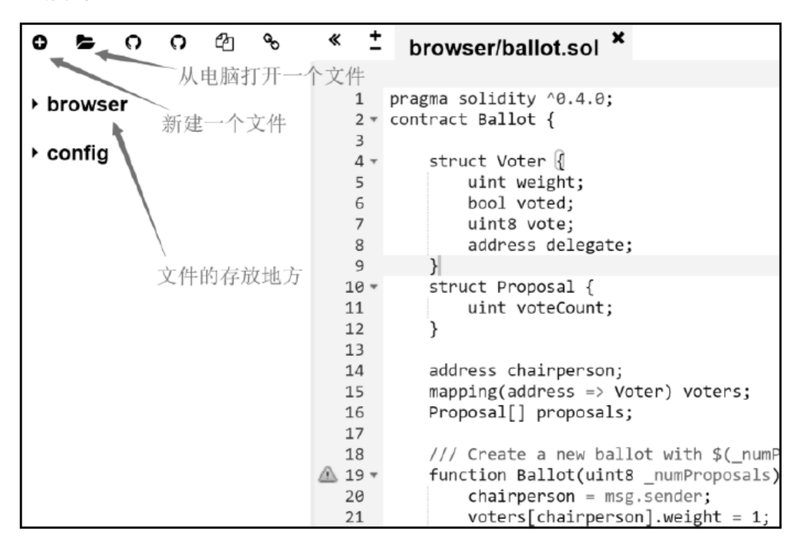



图 3-3　Remix 左上角的按钮工具栏

图 3-3 中间是 Solidity 代码编写区域，每一个 Solidity 代码文件的后缀是.sol，Remix 的功能非常丰富，在我们编写代码时，它会自动支持下面两个功能：

（1）语法关键字的自动提示功能（智能关键字提示功能），能够根据输入的首字母进行提示。

（2）具备自动编译功能，当在代码编写区域进行了修改时，Remix 默认会自动地对当前代码的.sol 文件进行编译，如果存在语法错误，会直接显示出来。

如图 3-4 所示，在 test.sol 文件中输入首字母 c 后，Remix 于是显示出了字母 c 开头的关键字提示，这些都是 Solidity 语法所支持的，此外由于"Auto compile"选项被勾选，因此还能自动进行编译，如果不想自动进行编译，把"√"去掉即可。




图 3-4　Remix 的智能关键字提示功能

在编译的时候，还可以选择编译器的版本。注意，不同的编译器版本编译出的 Bytecode（字节码）是不一样的，所支持的 Solidity 语法也不同。一般来说，选择带有“commit”单词且不含有“nightly”单词的编译器版本，这类版本的编译器不容易出问题。图 3-5 是一个选择编译器版本的例子。

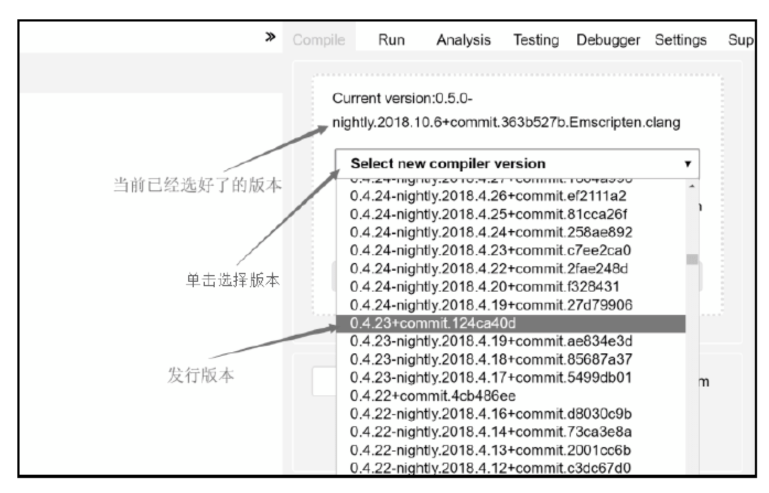



图 3-5　在 Remix 中选择合适的 Solidity 编译器版本

从图 3-5 可以看出，Solidity 编译器版本的名称比较复杂，这些名称符合下面的 3 个规则：

（1）版本号。
（2）前缀标记，通常是 develop.YYYY.MM.DD 或者 nightly.YYYY.MM.DD 。
（3）通过 commit.GitHash 提交。

例如，0.4.25+commit.59dbf8f1.Emscripten.clang，0.4.25 就是版本号，它的 git 提交的哈希值是 59dbf8f1，这是 Emscripten.clang 默认加上的。

此外，如果存在一些本地修改，提交时会自动加上.mod 后缀。

了解了 Remix 基础知识后，我们就能进行智能合约的编写了。

# 3.3　实现加法程序

下面使用 Solidity 编写一个实现加法计算的智能合约。没错！就简单地实现这么一个功能，它也是一份智能合约，一定要清楚并不是编写和代币相关的程序才叫智能合约。

单击 Remix 编译器左上角的“+”按钮，创建一个新的.sol 文件，名称为“test.sol”，如图 3-6 所示。

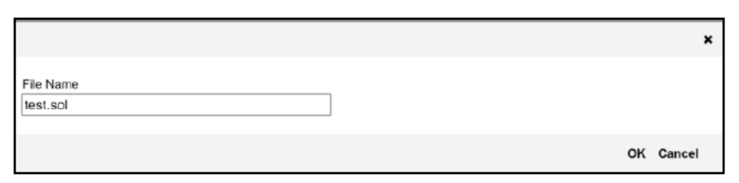



图 3-6　创建新文件

然后输入下面的代码：

```
pragma solidity ^0.4.23; //指定版本
contract Test {
    //  输入两个参数
    function add(uint8 arg1,uint8 arg2) public pure returns (uint8) {
        return arg1+arg2;
    }
}
```

如图 3-7 所示为编译器中的代码示例。

图 3-7 编译器中的代码示例

自动编译后的结果区域显示为绿色，证明代码在编译层面是没有问题的，如图 3-8 所示。

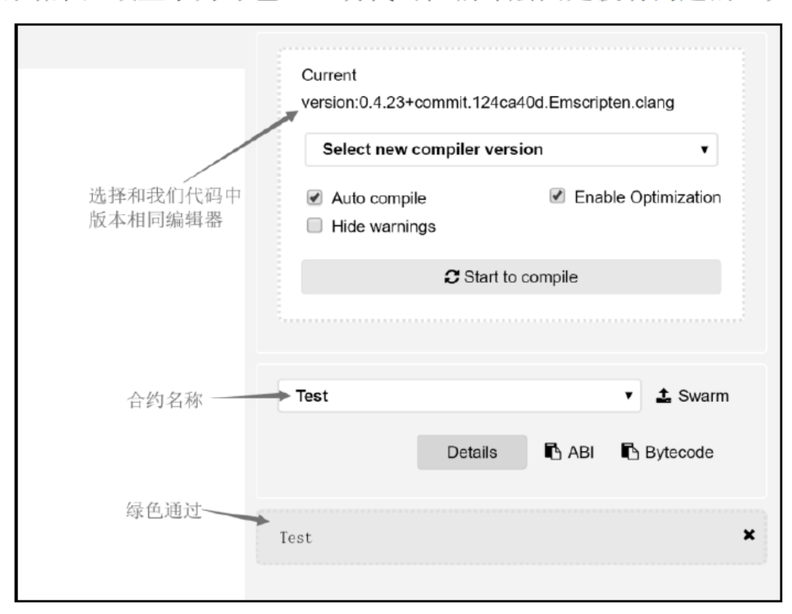



图 3-8 自动编译后的结果

以上就是一个简单的加法运算的智能合约的实现，我们稍后在下面的一节中对这份智能合约进行发布并调用。

# 3.4　实现 ERC20 代币智能合约

以太坊上发行的“代币”百分之九十以上都是基于“ERC20”标准的代币合约，可以说“ERC20”标准的代币合约目前是最为流行的合约标准。

本节我们来学习实现一个符合“ERC20”标准的代币智能合约，模仿上一节的步骤，在 Remix 编译器中新建一个名称为“MyToken.sol”的文件。

## 3.4.1　定义标准变量

首先在代码中定义标准要求的变量，如下所示：

```
pragma solidity ^0.4.23;     //指定版本
contract MyToken {
    string public name = "My first token coin";    // 代币的名称
    uint8 public decimals = 18;     // 代币单位精确到小数点后的位数
    string public symbol = "MFTC";  // 代币的符号
    uint public totalSupply = 100;   // 代币的发行量
}
```

如图 3-9 所示为编译器的代码示例。

```
browser/MyToken.sol
1  pragma solidity ^0.4.23; //指定版本
2
3
4  contract MyToken {
5
6      string public name = "My first token coin";    // 代币的名称
7
8      uint8 public decimals = 18;     //  代币单位精确到小数点后的位数
9
10     string public symbol = "MFTC";  //  代币的符号
11
12     uint public totalSupply = 100;  //  代币的发行量
13
14 }
15
16
17
```

图 3-9　编译器中的代码示例

这里我们定义了 name、decimals、symbol、totalSupply 共 4 个变量，注意，“totalSupply”变量表示发行量，如果这个发行量是一个很大的数，例如几百亿，那么最好将其变量类型设为 256 位的无符号整数类型，即“uint256”，以避免超过最大整数范围的上限。

## 3.4.2　事件与构造函数

在基础的变量定义好之后，我们定义合约中的“Event”事件，再补充一个构造函数，在构造函数中可以进行一些变量的初始化，例如把发行量“totalSupply”的初始化放入构造函数中，如图 3-10 所示。

```
ContractDefinition MyToken      0 refe
1  pragma solidity ^0.4.23; //指定版本
2
3
4  contract MyToken {
5
6      string public name = "My first token coin";    // 代币的名称
7
8      uint8 public decimals = 18;     //  代币单位精确到小数点后的位数
9
10     string public symbol = "MFTC";  //  代币的符号
11
12     uint public totalSupply;  //  代币的发行量
13
14     //  下面是转账Event 和 授权Event
15     event Transfer(address indexed _from, address indexed _to, uint256 _value);
16     event Approval(address indexed _owner, address indexed _spender, uint256 _value);
17
18     // constructor 是 Solidity 构造函数的关键词
19     constructor() public {
20         totalSupply = 100; //  发行量初始化
21     }
22
23 }
24
```

图 3-10　将 totalSupply 的初始化放入构造函数

## 3.4.3　Solidity 的常见关键字

在上一节的构造函数代码中，可以看到“constructor”后面有一个“public”，这个 public 是 Solidity 语言的一个关键字。除了这个关键字之外，Solidity 语言还有下面 6 个关键字，了解这些关键字的含义对于我们理解和编写智能合约很有必要。

- public，可以修饰变量和函数，被修饰的函数或变量可以被任何合约调用（或访问）。默认的变量和函数使用该属性。
- private，可以修饰变量和函数，被修饰者只能被当前合约内部的代码所调用（或访问），不能被外部合约调用或继承它的子合约调用（或访问）。
- external，只能修饰函数，被修饰的函数只能被当前合约之外的合约所调用（或访问），不能被自己和继承它的合约调用（或访问）。
- internal，可以修饰变量和函数，被修饰者可以被当前合约内部以及继承它的合约调用（或访问），但不能被外部合约调用（或访问）。
- view，只能修饰函数，函数内部能够对外部变量进行读取操作，但是不能进行修改。
- pure，只能修饰函数，函数内部不能对外部的变量进行读取和修改操作，它只能对传参进入的参数量进行读写操作。

下面我们根据源码示例来进一步理解这些关键在，请注意代码中的注释。

```
pragma solidity ^0.4.23; //指定版本

contract MyToken{  // 外部合约
    function getHalfTotalSupply() external view returns (uint half);
    function internalFunc() internal view returns (uint half);
}

contract parent {
    uint64 age = 50;
    // addr 是 MyToken 合约部署在链上后的地址
    address public addr = 0x72bA7d8E73Fe8Eb666Ea66babC8116a41bFb10e2;
    function func() public view {
        MyToken m = MyToken(addr); // 实例化外部合约
```

```
            m.getHalfTotalSupply(); // external 允许 parent 调用外部合约 MyToken 的函数
            m.internalFunc(); // 报错，因为这个是 internal 的函数（或方法），
                                  //只能在 MyToken 内部或继承了它的合约中使用
        }
        function publicFunc() public {
            age = age + 6;
        }
        function privateFunc() private returns (uint64 ret) {
            uint64 t = internalFunc();
            age = t / 2;
            return age;
        }
        function internalFunc() internal returns (uint64 ret) {
            age = age * 2;
            return ret;
        }
        function viewFunc(uint64 arg1) public view returns (uint64 ret) {
            arg1 = arg1 + age + 9; // 可以访问 age
            // age 初始值是 50
            age = age + 7;          // 尝试修改，编译会发出警告，但是编译可以通过
            uint64 d = age + arg1; // 变量 age 的值不会改变，依然是 50
            uint64 c = d/2;
            return c;
        }
        function pureFunc(uint64 arg1) public pure returns (uint64 ret) {
            arg1 = arg1 + 9;
            uint64 d = age + arg1; // 编译报错！pure 完全禁止外部 age 变量的读写
            //uint64 c = d/2;
            return arg1;
        }
    }

    contract child is parent {
        function usePrivateFunc() public returns (uint64 ret) {
            uint64 v = privateFunc(); // 报错，尝试调用 parent 合约中的 private 函数
            return v;
        }
        function useInternalFunc() public returns (uint64 ret) {
            uint64 v = internalFunc(); // 可以使用，因为 child 继承自 parent
            return v;
        }
    }
```

### 3.4.4　授权与余额

接着我们继续补充标准中的代币余额和授权额度。首先，代币余额和授权额度存储的数据结构是一个“Map”，“Key”对应的是以太坊的地址，然后“Value”对应的是数值，该数值根据用户的以太坊地址来映射用户所拥有的余额或额度的多少，查询的时间复杂度是 O(1)，这样的查询效率是很高的。相对应地，也要实现标准中的“balanceOf”代币余额查询、“approve”授权额度申请和“allowance”授权额度查询这 3 个函数，如图 3-11 所示。

```
18      mapping (address => uint256) public balances; //  余额 Map
19      mapping (address => mapping (address => uint256)) public allowed; //  授权Map
20
21      // constructor 是 Solidity 构造函数的关键词
22      constructor() public {
23          totalSupply = 100; //  发行量初始化
24          balances[msg.sender] = totalSupply;
25      }
26
27      // 根据地址获取代币余额
28      function balanceOf(address _owner) public view returns (uint256 balance) {
29          return balances[_owner];
30      }
31
32      //  授权额度申请
33      function approve(address _spender, uint256 _value) public returns (bool success) {
34          allowed[msg.sender][_spender] = _value;
35          emit Approval(msg.sender, _spender, _value);
36          return true;
37      }
38
39      // 根据 _owner 和 _spender 查询 _owner 给 _spender 授权了多少额度
40      function allowance(address _owner, address _spender) public view returns (uint256 remaining) {
41          return allowed[_owner][_spender];
42      }
43
```

图 3-11　ERC20 标准的部分函数

在“approve”函数中，内部代码倒数第二行“emit”触发的就是授权事件“approve event”。

定义好余额相关的操作后，如果我们想在合约创建的时候就对某一个账号进行代币数值的赋值，可以在构造函数“constructor”中进行，这也是一般的合约发布把代币转出到一个对公的以太坊地址的做法，之所以这样做是因为智能合约拥有一个和钱包地址格式完全一样的以太坊地址，但是在合约创建的时候，我们没有这个智能合约的私钥，也就是说，我们不能直接像导入钱包一样将合约导入到某个支持 ERC20 代币转账的应用中，把代币转出。

还有一点，如果我们要实现智能合约的代币数值锁仓，也可以在合约中通过自己实现的函数来达到目的。下面修改构造函数中的代码，对合约发布者进行代币赋值，如图 3-12 所示。

```
13      // constructor 是 Solidity 构造函数的关键词
14      constructor() public {
15          totalSupply = 100000000000000000000000000; //  修改发行量初始一亿
16          balances[msg.sender] = totalSupply;
17      }
```

图 3-12　在构造函数初始化发行量并将所有的代币给予某个地址

其中的“msg.sender”就是当前合约发布者的以太坊地址，我们把“totalSupply”发行量全部给予了这个地址。当然，也可以进行控制，不全部给予，只给一小部分，这些都是允许的。

## 3.4.5　转账函数

下面在合约中完成标准中的转账函数“transfer”，如图 3-13 所示。

```
34      // 代币转账
35      function transfer(address _to, uint256 _value) public returns (bool success) {
36          require(_to != 0x0);
37          require(balances[msg.sender] >= _value);
38          require(balances[_to] + _value >= balances[_to]);
39          balances[msg.sender] -= _value;
40          balances[_to] += _value;
41          emit Transfer(msg.sender, _to, _value);
42          return true;
43      }
44
```

图 3-13　完成转账函数“transfer”

转账函数的内部首先要进行一些参数校验，一般使用 Solidity 语法中的“require”关键字来进

行限制。例如，在上面的代码中，“transfer”中的第一行限制收款地址不能是零地址，第二行对代币的转出方“msg.sender”所持有的余额数值和想要被转出的数值“value”进行比较判断，判断余额是否比“value”多，因为要转出，所以余额必须比“value”多。

在参数判断通过后，通过减少转出者的余额值，即“balances[msg.sender]-=value”，来减去“value”，然后增加收款者的余额，即“balances[_to]+=value”这么一个过程来实现数值层面的转账。这里要注意，到“balances[_to]+=value”这一行，目前的修改都是基于内存层面的，还没有写到区块链上，也就是说这些修改还没有真实地在区块链上生效。

最后，我们要发出一个转账事件“transfer event”，然后返回 true。返回 true 的时候，如果当前调用的“transfer”函数由矿工提取“sendRawTransaction”中的 data 传入 EVM，那么转账的修改就会被持久地记录下来，并真实生效。

还可以使用另外一个函数“transferFrom”来实现转账，具体的代码如图 3-14 所示。

```
32      // 代币转账
33      function transfer(address _to, uint256 _value) public returns (bool success) {
34          require(_to != 0x0);  // 不允许收款地址是零地址
35          require(balances[msg.sender] >= _value);
36          require(balances[_to] + _value > balances[_to]);
37          balances[msg.sender] -= _value;
38          balances[_to] += _value;
39          emit Transfer(msg.sender, _to, _value);
40          return true;
41      }
42      // 代币转账2
43      function transferFrom(address _from, address _to, uint256 _value) public returns (bool success) {
44          require(_to != 0x0);
45          uint256 allowanceValue = allowed[_from][msg.sender];
46          require(balances[_from] >= _value && allowanceValue >= _value);
47          require(balances[_to] + _value > balances[_to]);
48          allowed[_from][msg.sender] -= _value;
49          balances[_to] += _value;
50          balances[_from] -= _value;
51          emit Transfer(_from, _to, _value);
52          return true;
53      }
```

图 3-14　“transfer”和“transferFrom”函数

因为“transferFrom”的转账方式涉及授权值，所以在参数判断阶段必须多出一个授权值“allowanceValue”和所要转出数值“value”的判断，转出的数值“value”必须比已经授权了的值小。

授权值“allowanceValue”的设置在“approve”函数中，关于参数的对应获取关系，在“标准的函数”一节介绍“approve”函数时举了一个很通俗的例子，以供读者参考。

在参数判断通过后，还要进行同“transfer”函数中一样的余额修改操作，即要减少当前的授权余额值。

至此，一份标准的 ERC20 代币的智能合约就编写完成了，其完整代码如下：

```
    pragma solidity ^0.4.23; //指定版本
    contract MyToken {
        string public name = "My first token coin";    // 代币的名称
        uint8 public decimals = 18;     //  代币单位精确到小数点后的位数
        string public symbol = "MFTC";  //  代币的符号
        uint public totalSupply;  //  代币的发行量
        // 下面是转账事件和授权事件
        event Transfer(address indexed _from, address indexed _to, uint256 _value);
        event Approval(address indexed _owner, address indexed _spender, uint256
_value);
```

```
        mapping (address => uint256) public balances; // 余额映射
        mapping (address => mapping (address => uint256)) public allowed;
        // 授权映射
        // constructor 是 Solidity 构造函数的关键字
        constructor() public {
            totalSupply = 100000000000000000000000000; // 修改发行量初始一亿
            balances[msg.sender] = totalSupply;
        }
        // 根据地址获取代币余额
        function balanceOf(address _owner) public view returns (uint256 balance)
{
            return balances[_owner];
        }
        //  授权额度申请
        function approve(address _spender, uint256 _value) public returns (bool
success) {
            allowed[msg.sender][_spender] = _value;
            emit Approval(msg.sender, _spender, _value);
            return true;
        }
        // 根据_owner 和_spender 查询_owner 给_spender 授权了多少额度
        function allowance(address _owner, address _spender) public view returns
(uint256 remaining) {
            return allowed[_owner][_spender];
        }
        // 代币转账
        function transfer(address _to, uint256 _value) public returns (bool success)
{
            require(_to != 0x0);  // 不允许收款地址是零地址
            require(balances[msg.sender] >= _value);
            require(balances[_to] + _value > balances[_to]);
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            emit Transfer(msg.sender, _to, _value);
            return true;
        }
        // 代币转账 2
        function transferFrom(address _from, address _to, uint256 _value) public
returns (bool success) {
            require(_to != 0x0);
            uint256 allowanceValue = allowed[_from][msg.sender];
            require(balances[_from] >= _value && allowanceValue >= _value);
            require(balances[_to] + _value > balances[_to]);
    allowed[_from][msg.sender] -= _value;
            balances[_to] += _value;
            balances[_from] -= _value;
            emit Transfer(_from, _to, _value);
            return true;
        }
    }
```

在这份“MyToken.sol”智能合约中，所有实现了的函数都是 ERC20 标准中的函数，我们还可以在里面添加一些其他的函数，例如添加一个返回总发行量一半的函数，如图 3-15 所示。

```
49  }
50
51      // 返回总发行量的一半
52      function getHalfTotalSupply() public view returns (uint half) {
53          return totalSupply/2;
54      }
55
56  }
57
58
59
```

图 3-15　在遵循合约标准的代码中自定义函数

在合约编写和编译好之后，发布阶段还要从 Remix 中取出合约的 Solidity 代码、合约的 ABI 和合约的字节码（Bytecode），以便于在调用合约函数时作为参数来传参。除了从 Remix 中取出合约的 Solidity 代码外，后两者的数据会在下面与合约的发布相关的章节来进行详解。

## 3.4.6　合约的代码安全

因为智能合约是由代码编写的，自然就会存在代码漏洞方面的风险，最著名的因为合约代码存在计算溢出漏洞而导致资产损失惨重的例子就是 2018 年 4 月份中的“美链 BEC 合约漏洞事件”，该事件造成的后果是“BEC”代币价值归零，其漏洞代码如图 3-16 所示。

```
254
255   function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) {
256     uint cnt = _receivers.length;
257     uint256 amount = uint256(cnt) * _value;
258     require(cnt > 0 && cnt <= 20);
259     require(_value > 0 && balances[msg.sender] >= amount);
260
261     balances[msg.sender] = balances[msg.sender].sub(amount);
262     for (uint i = 0; i < cnt; i++) {
263         balances[_receivers[i]] = balances[_receivers[i]].add(_value);
264         Transfer(msg.sender, _receivers[i], _value);
265     }
266     return true;
267   }
268 }
269
```

图 3-16　有漏洞的合约代码

“batchTransfer”是一个实现批量转账的函数，图 3-16 中框选部分就是有问题的代码，即

```
uint256 amount = uint256(cnt) * _value;
```

其中，“_value”是一个类型为“uint256”的入参，当传入的“_value”很大且恰好还没有超过“uint256”最大可表示的数值范围而接近“uint256”取值范围的最大值时，“_value”乘上“uint256(cnt)”最终超过范围，造成“uint256”数据类型的溢出，此时“amount”就变成一个很小的数，不再是正确的值，这将使得后面的判断条件很容易被校验通过。

当从转币者的地址扣除“amount”的时候，事实是扣了很少。

```
balances[msg.sender] = balances[msg.sender].sub(amount);
```

而在循环中给“_receivers”添加代币数量的时候，却添加了“_value”的数值。

```
balances[_receivers[i]] = balances[_receivers[i]].add(_value);
```

最终导致扣除了很少的“amount”却转给别人巨额的数值，造成资产被盗！

解决这个问题的方法是使用安全的乘法方式，以避免大数溢出，例如使用如下的乘法函数：

```
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
            return 0;
        }
        uint256 c = a * b;
        require(c / a == b);
        return c;
    }
```

对于智能合约中的加、减、乘、除运算，以太坊官方提供了一个安全运算函数的开源库，链接如下：

https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol

以上只是合约代码安全方面的例子之一。在实际开发中，在复杂且涉及运算的功能实现中一定要谨慎编写代码，以避免产生代码漏洞。

# 3.5 链上的合约

发布智能合约又称把智能合约发布到区块链上。智能合约一旦发布成功，便不能再修改，这是一个不可逆的操作。所谓的不能修改，指的是合约的代码不能重新编写。

一般来说，在把编写好的智能合约真正发布到公链之前，会先将其发布到以太坊测试网络的测试链上，进行广泛的测试，包含 bug 点检测等，确保没问题才会发布到公链上。

如果智能合约发布到公链后发现依然存在问题，怎么办呢？这种情况下只能重新发布一份替换的智能合约，并将之前发布的有问题的智能合约宣布作废，新发布的合约的名称此时就会出现和旧合约的名称一样的情况，根据合约的唯一性标志，名称一致没有关系，合约的以太坊地址总是不一样的。图 3-17 所示就是在以太坊区块链浏览器中输入 ERC20 代币名称进行查询的时候，看到的名称相同的代币列表。

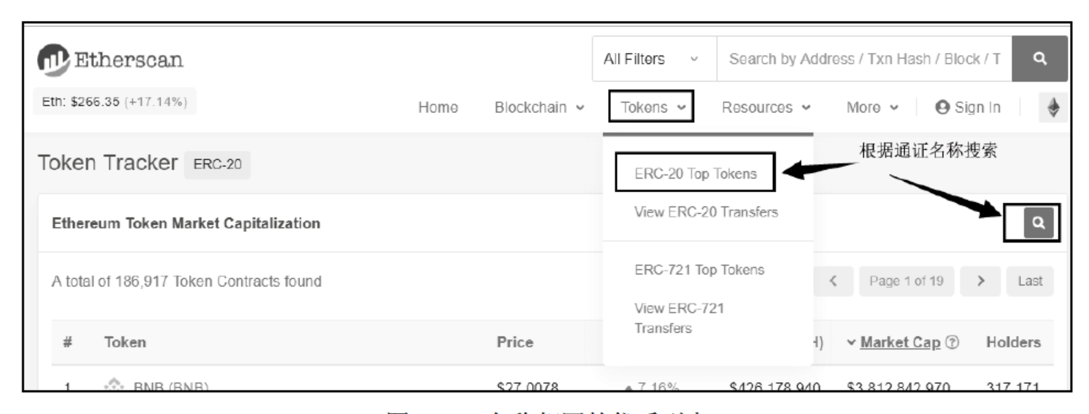



图 3-17　名称相同的代币列表

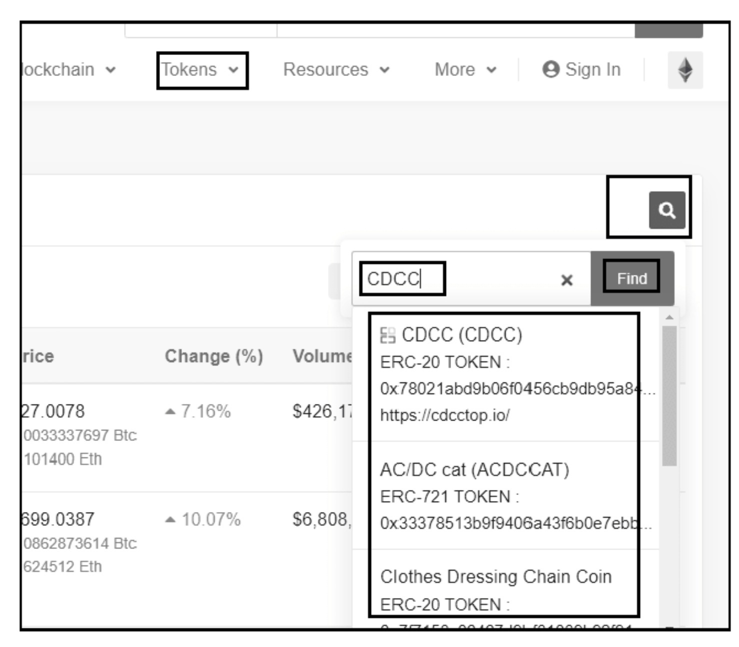



图 3-17（续）

如果还涉及问题合约对应的代币已经流通起来了的情况，此时的补救方法可以考虑：选择一个区块高度，以所选区块高度为准，对旧合约代币的所有持有者（Holder）进行新合约代币数值的一一对应的映射操作，再宣布旧合约地址作废，合约以新的地址为准。

## 3.6　认识 Mist

和智能合约的编译器 Remix 一样，以太坊也提供了一个包含智能合约发布功能的图形界面钱包，名称是 Mist，也可称为“Ethereum Wallet”（以太坊钱包）。Mist 是一个开源软件，并支持 Windows（64 位）、Mac 、Linux 三大操作系统，它的源码开源地址是 https://github.com/ethereum/mist。

Mist 安装包的下载链接是：https://github.com/ethereum/mist/releases。如图 3-18 所示。

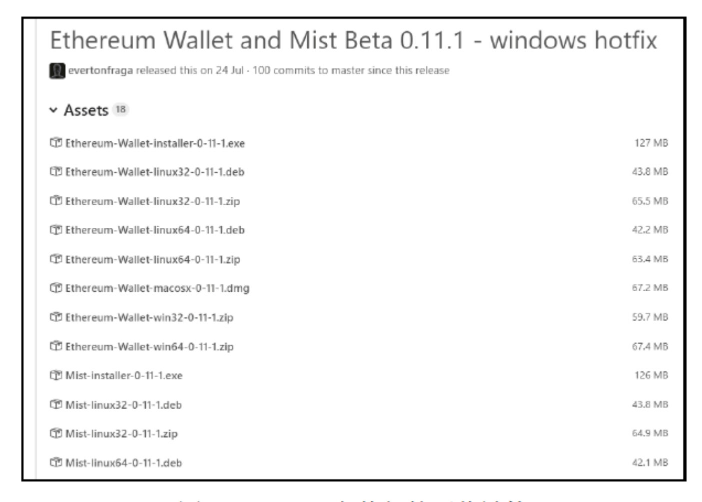



图 3-18　Mist 安装包的下载链接

Mist 除了具有发布智能合约的功能外，主要还是一个以太坊钱包，自然会拥有钱包的创建、备份以及以太坊 ETH 余额查看、ETH 转账和 ETH 挖矿的功能。

下面以 Windows 64 位的 Mist 版本为例介绍其启动方式，首先下载 zip 压缩包，解压缩到电脑中的某一个文件夹下即可，如图 3-19 所示。

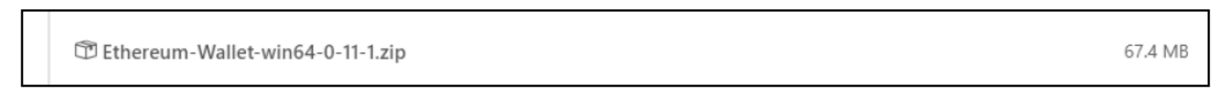



图 3-19　要下载的 Mist 版本

随后打开解压缩好的文件夹，然后单击“Ethereum Wallet.exe”启动以太坊钱包软件，如图 3-20 所示。

| | | | |
|---|---|---|---|
| d3dcompiler_47.dll | 2018/3/16 17:52 | 应用程序扩展 | 4,077 KB |
| Ethereum Wallet.exe | 2018/7/23 16:27 | 应用程序 | 65,885 KB |
| ffmpeg.dll | 2018/3/16 17:47 | 应用程序扩展 | 1,899 KB |
| icudtl.dat | 2018/3/16 17:47 | DAT 文件 | 9,894 KB |

图 3-20　启动以太坊钱包软件

首次启动会有点慢，稍等一会儿就能进入到主页面，如图 3-21 所示。

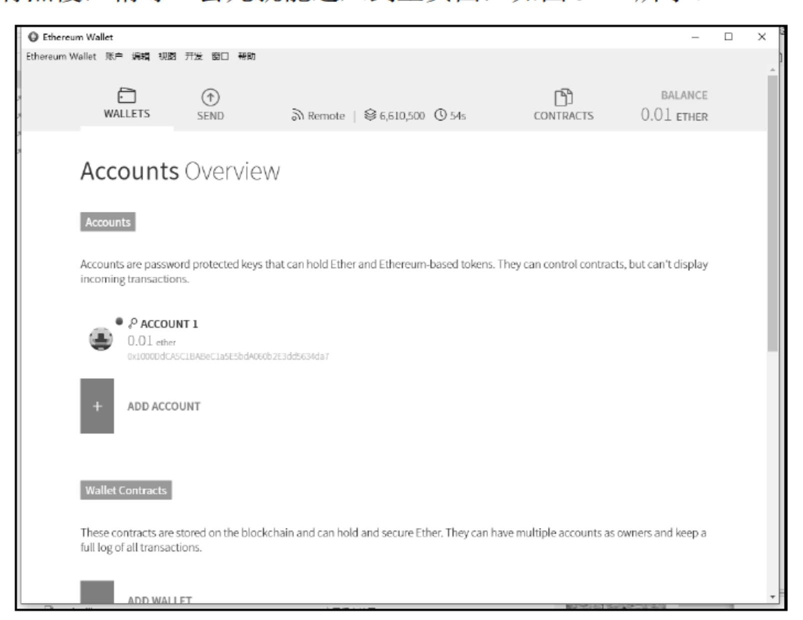



图 3-21　以太坊钱包的主页面

下面我们开始介绍 Mist 的主要功能及使用。

## 3.6.1　节点的切换

在“Mist”主页左上角的功能按钮栏中，我们可以设置当前我们想要连接的以太坊网络节点，切换方式为用鼠标依次单击“开发”→“网络”。

在所展示的列表中，“主网络”代表的是以太坊主网，在主网站进行的转账操作发生的资产

变化都是具有真实法币价值的。“Ropsten”网络代表的是以太坊的测试网络之一，顾名思义，测试网络中所产生的资产操作都属于测试性质，不被承认具有真实法币价值。所谓法币价值，就是我们日常所使用的法定货币（金钱）的价值。“Rinkeby”也是以太坊测试网络中的一类，它与“Ropsten”的区别会在后面的“获取测试网络节点”一节中进行讲解。

### 3.6.2 区块的同步方式

和“节点的切换”一样，在“Mist”主页左上角的功能按钮栏中也可以切换同步网络类型链上区块的同步方式，这是什么意思呢？

原来是“Mist”软件在选择好以太坊网络类型后会自动同步该网络区块链上的区块，将区块存放在本地电脑，以方便软件自身从区块中读取数据。

因为区块链在运行的过程中，伴随着区块的不断产生，链上的区块越来越多，比如目前以太坊公链区块数量已经达到了 600 多万个，面对如此多的区块数据量，如果软件完整地从 0~600 多万区块同步到我们的计算机，这将会非常耗时。因此，以太坊的节点源码提供了可以选择同步方式的接口，以方便开发者根据需要作出选择。

执行“开发”→“Sync mode”，在可选的列表中提供了以下同步方式：

- Light 模式。轻节点模式，策略是只同步所有区块的头部信息，区块体不同步下载。
- Fast 模式。快速模式，策略是快速同步完所有区块的头部信息，随后根据每个区块头同步对应的区块体，最终达到完全同步的目的。
- Full 模式。全节点模式，这个模式是最耗时的，它将直接从区块头开始完整同步区块信息。
- Mist 默认选择的是 Light 模式。

## 3.7 创建以太坊钱包

以太坊智能合约的发布，存在着一个“Creator”（创建者）的概念，这个“Creator”的意思是指这份智能合约是由哪个以太坊地址发布的。从交易的角度来看，以太坊上每份合约的发布本质上都是发送一笔交易，即“Transaction”。

显然，有交易就有交易发起者，而发布智能合约交易的发起者就是“Creator”，在合约发布的时候，这个地址对应 Solidity 代码里面的“msg.Sender”，因此要求当前“Creator”的以太坊地址拥有以太坊 ETH 代币，以便在发布合约时作为交易的手续费。

首先我们跟随图 3-22 在 Mist 中创建一个以太坊钱包来充当发布智能合约的“Creator”。单击右上角功能列表中的“账户”，再单击新建账户。

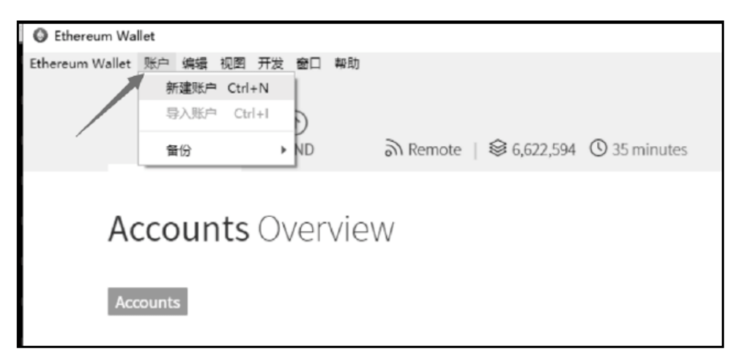



图 3-22　在 Mist 中创建以太坊钱包

在弹出的对话框中，输入密码即可完成账户的创建，如图 3-23 所示。




图 3-23　输入创建钱包的密码

在弹出的英文提示框中提示如何备份在 Mist 中新建钱包的方法，如图 3-24 所示。

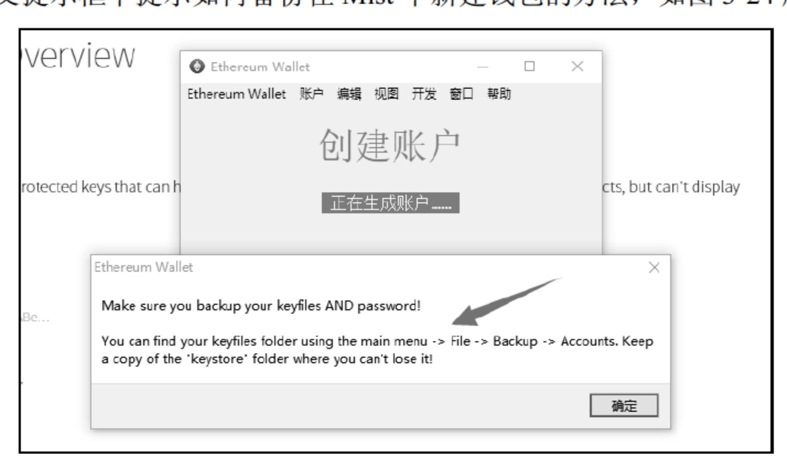



图 3-24　提示信息

备份钱包的流程是：依次单击“账户”→“备份”→“账户”。按照此流程操作后，可以看到使用 Mist 生成的钱包的“keystore”文件所存放的文件夹，知道了这个文件夹，就能把钱包的“keystore”文件取出，之后可将其导入到其他的钱包软件 App 中，也可以在开发时使用，如图 3-25 所示。

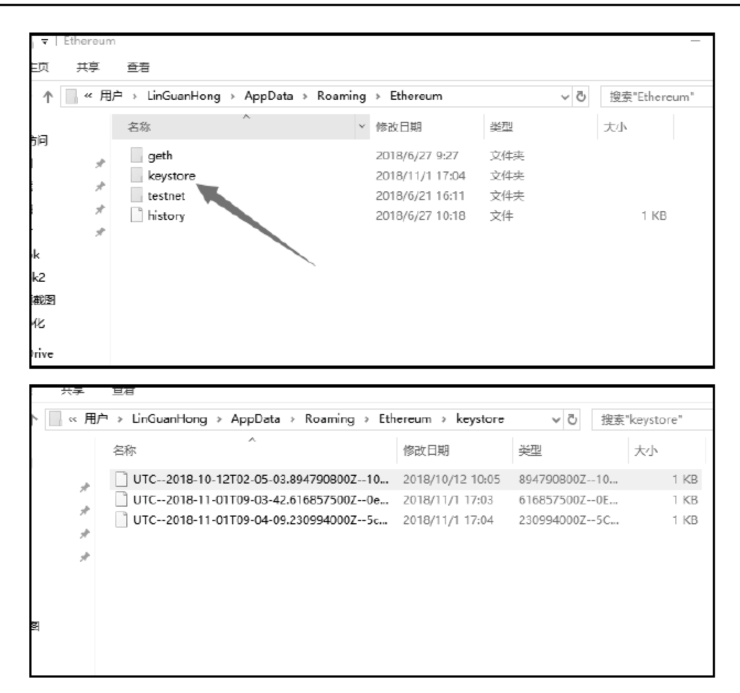



图 3-25　Mist 创建的钱包所生成的 Keystore 文件

图 3-25 中的每一个“keystore”文件都对应一个以太坊钱包地址，那么如何区分各个地址呢？其实很容易。例如，图 3-25 中有 3 个以太坊钱包的“keystore”文件，只需要先观察每个文件的名称再和我们所知道的钱包地址进行匹配就能区分。当然，除了借助文件名称来识别，还可以直接以文本格式打开“keystore”文件，在文件里面也能看到钱包地址，也就是去掉了 0x 后的所有字符，如图 3-26 所示。

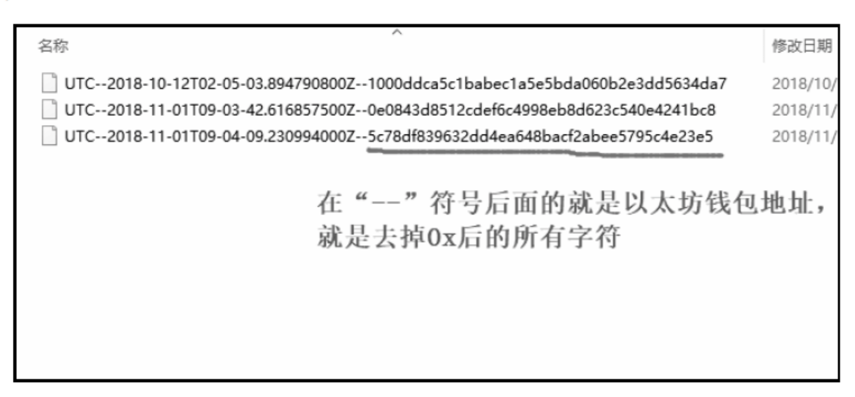



图 3-26　Mist 默认给钱包 keystore 文件的命名方式

拥有了“keystore”文件还不足以让我们解锁这个钱包，在使用“keystore”文件解锁钱包，例如导入到钱包 App 的时候，还需要输入这个“keystore”文件对应的密码，这个密码就是我们在创建钱包时输入的密码。

“keystore”文件及其密码和钱包私钥、助记词的关系如下：

“keystore”文件+密码=私钥

“keystore”文件+密码=助记词

最后，创建完成的钱包界面如图 3-27 所示。每个钱包所对应的“x.xx ether”代表的就是拥有

多少个以太坊 ETH 代币。需要注意的是，Mist 显示的数值只精确到了小数点后两位且最后一位小数位是四舍五入形式的。例如，如果刚好有 0.001 Ether，那么在这里看到的是 0.00 Ether，如果刚好有 0.0051 Ether，那么在这里看到的应该是 0.01 Ether。

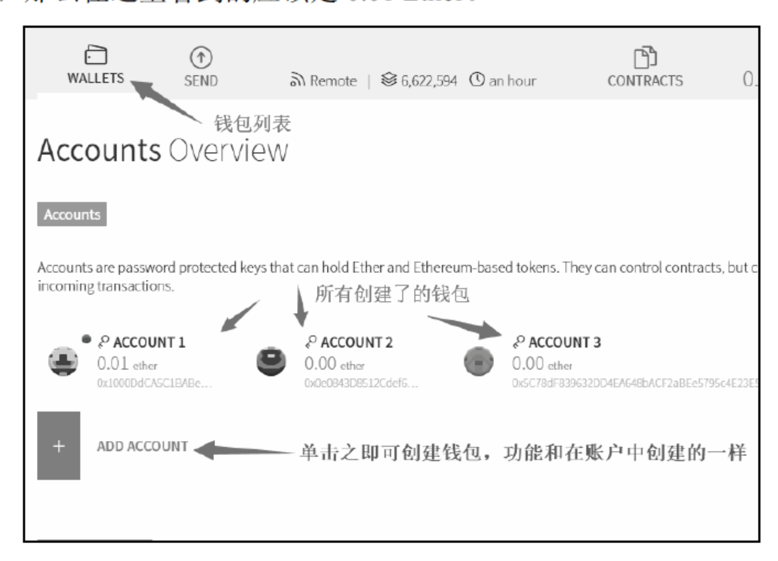



图 3-27 Mist 钱包列表界面

## 3.8 使用 Mist 转账代币

创建好的以太坊钱包，除了用来发布智能合约，还能用来接收别人转给我们的以太坊 ETH 或 ERC20 代币。

转账也是 Mist 支持的一项功能，在主页面中单击“SEND”按钮进入交易页面，注意，这个页面既可以进行以太坊 ETH 转账，又可以进行 ERC20 代币转账。如图 3-28 所示。




图 3-28 Mist 的转账界面

图 3-28 中的“TO”代表的就是收款人的以太坊钱包地址，必须满足十六进制共 42 个字符的格式，否则会出错。可选的能够进行转账的代币列表中默认的第一项是以太坊 ETH，也就是图中

的“ETHER”，可以通过用鼠标单击来选择。

除了“ETHER”之外的选项都是其他的代币，这里的代币不局限于 ERC20 代币类。注意，选择列表中显示的其他代币，需要满足下面的两个条件：

（1）代币对应的合约必须是用户，也就是我们在 Mist 中手动添加的“Custom Tokens”（定制代币），可以单击主页面中的“CONTRACTS”按钮，再下滑到最后，单击“WATCH TOKEN”按钮，手动添加代币，如图 3-29 所示。

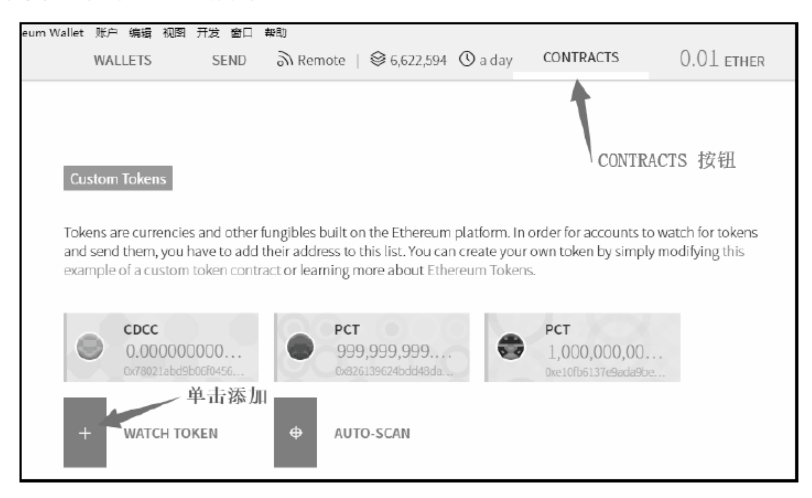



图 3-29 Mist 中添加进来的合约界面

（2）必须是当前在 Mist 解锁了的钱包账号，该钱包拥有数量大于零的代币。

说明一下第（2）点，解锁了的主钱包就是图 3-29 名称为“Account 1”的钱包，“Account 1”必须拥有图 3-29 中两个“PCT”代币的值。这样就满足了第（2）个条件所描述的要求，如图 3-30 所示。

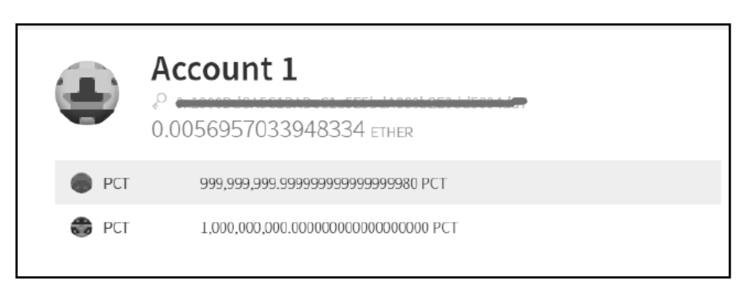



图 3-30 两个 PCT 代币值

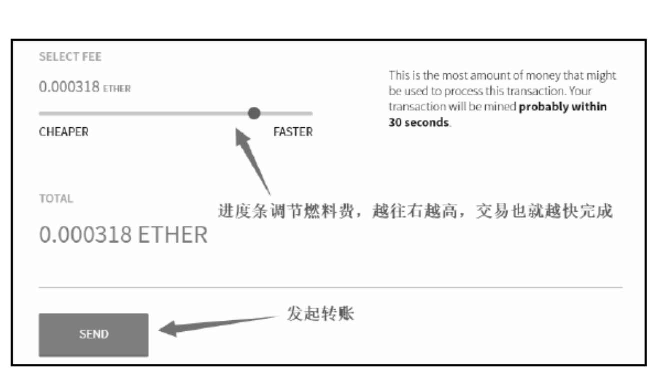
仍然在交易页面单击“SHOW MORE OPTIONS”按钮，可以看到下面的“DATA”输入框（见图 3-31），这里的“数据”是可选输入的，它具有下面的特点：

- 必须是 ETH 转账的情况，其他代币转账，Mist 是不允许输入的。
- 必须是十六进制格式的字符串，否则报错。
- 这里的“数据”在代码层面对应的就是我们在“以太坊交易”一节中介绍的“data”参数。




图 3-31　Mist 在 ETH 交易情况下提供的 DATA 参数输入框

图 3-32 是交易手续费的调节界面和转账发送最后一步等待输入解锁密码的界面。



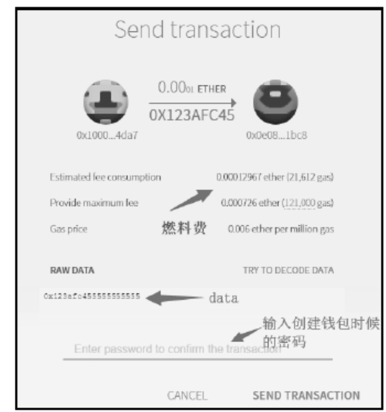
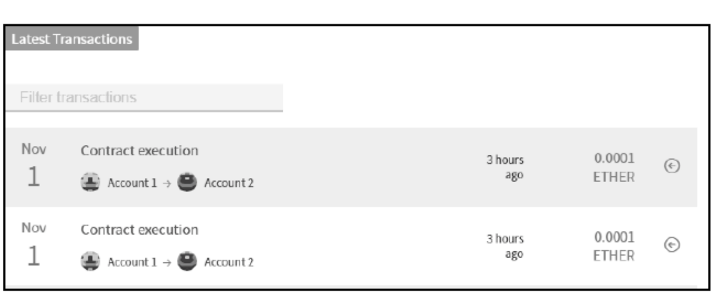




图 3-32　交易手续费的调节和输入解锁密码的界面

输入密码后按回车键，就能进行交易了。交易发起之后，回到 Mist 主页面，滑动鼠标到当前页面的后面就能看到每笔已经发起了的交易记录，单击交易记录即可查看交易详情，如图 3-33 所示。



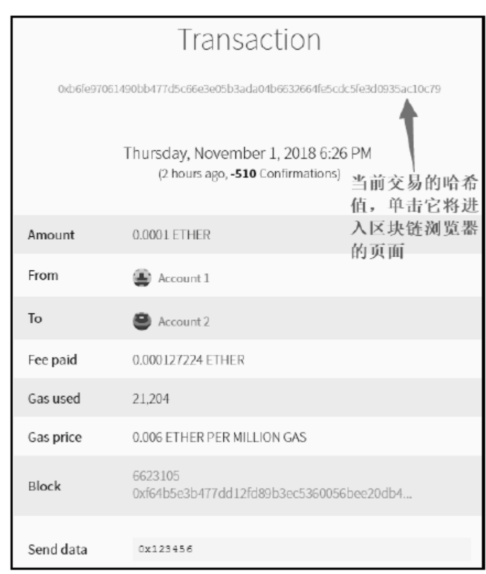




图 3-33　Mist 中显示已经发送了的交易信息界面

现在使用前面“区块链浏览器”一节中所介绍的“etherscan.io”网址来查询刚刚测试中发起的ETH 转账，即图 3-32 中“data”设置为“0x123afc455555555555”的那一笔。如图 3-34 所示，可以看到交易已经成功，而且数据都是我们所设置的那样。

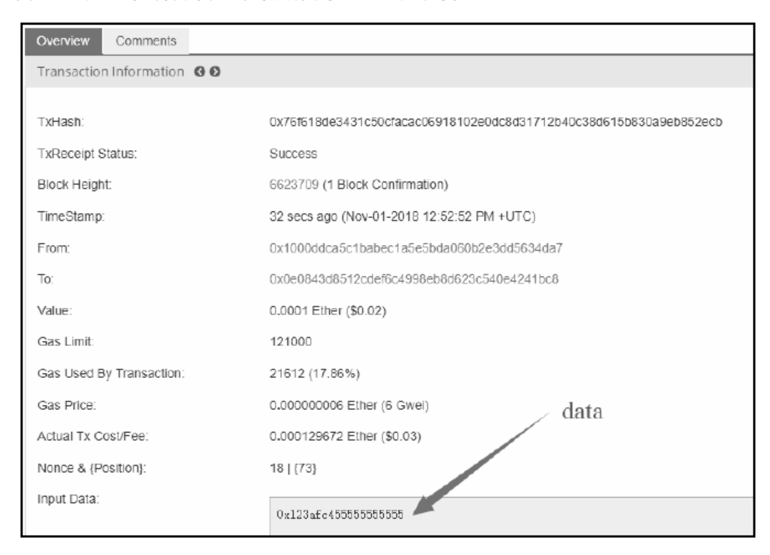



图 3-34　在区块链浏览器中查看交易的 data 参数

由“data”在以太坊 ETH 转账结果的体现来看，我们可以利用“data”来自定义输入自己想要记录的数据，然后利用交易将这些数据发送到区块链上，这样“data”中的数据就会永远存在于区块链上。

# 3.9　使用 Mist 发布智能合约

首先在 Mist 的主页面单击“CONTRACTS”按钮，在显示的主页面中单击“DEPLOY NEW CONTRACT”按钮就可以开始合约的部署，如图 3-35 所示。“DEPLOY NEW CONTRACT”的中文含义就是部署新合约。

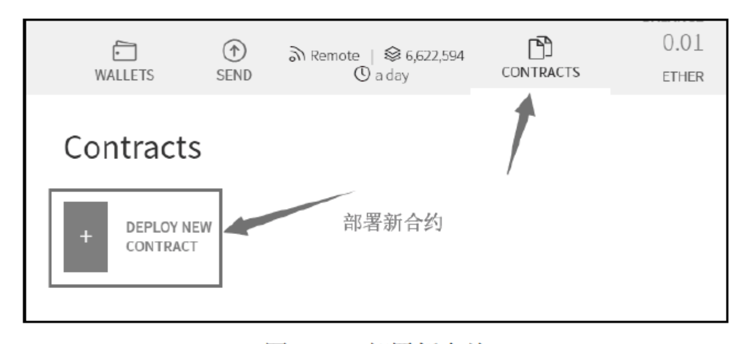



图 3-35　部署新合约

图 3-36 表示的意思是，在部署合约的同时发送多少个 ETH 代币到这个合约的地址，一般保持 0 个即可，因为目前所有发送到某个智能合约地址上的 ETH 代币都是拿不回来的。智能合约的以太坊地址虽然和钱包地址格式一样，但在部署合约成功后是不会获得当前合约地址所对应的私钥或

者助记词的。




图 3-36 Mist 发送 ETH 交易的默认数值

在当前页面继续往下滑动，可以看到输入合约 Solidity 代码的编辑框，以及输入合约“Byte Code”的地方，与此同时，Mist 还提供了对这两个编辑框中的内容进行校验的功能，例如语法校验，如图 3-37 所示。

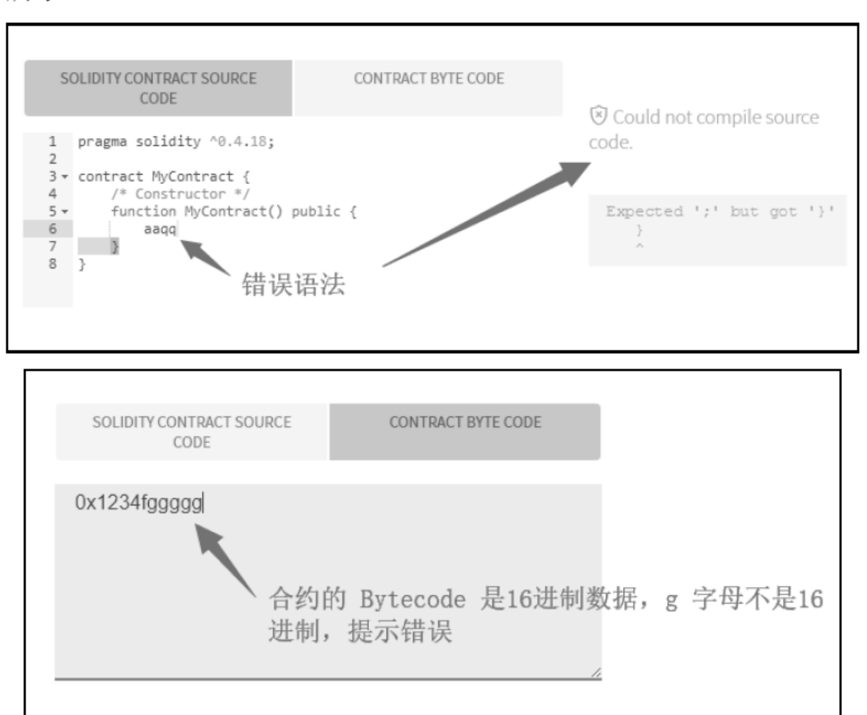



图 3-37 Mist 的 Solidity 代码错误语法的提示

虽然 Mist 为开发者提供了使用 Solidity 语法编写智能合约的功能，但是笔者并不推荐这种做法，建议使用 Remix 进行智能合约的编写，再从 Remix 中提取出相应的内容，复制到 Mist 上述两个输入框中，然后进行合约的部署。

此时，需要提取出在 Mist 发布合约用到的核心信息，包括合约的 Solidity 代码、合约的 ABI 和合约的 Bytecode（字节码）。下面介绍这些信息的提取方法。

### 3.9.1　合约 Solidity 源码

仍以“实现加法程序”一节中的加法智能合约为例。在编译通过后，如果要使用 Solidity 的代码，可直接从 Remix 的编辑框中提取，提取后再粘贴到 Mist 的 Solidity 代码输入框中，如图 3-38 所示。

这种直接复制 Solidity 代码的操作是最简单的，但也存在问题，特别是对于代码量多、复杂度高的智能合约，不建议使用直接复制的方式编译发布。问题是这样的：在 Remix 中按照 ERC20 标准编写好了 Solidity 后，复制粘贴到 Mist 发布，发布成功后，到区块链浏览器 https://etherscan.io/ 中查看刚发布成功的合约，会发现少了“Read Contract”文字按钮，如图 3-39 所示。

browser/MyToken.sol　browser/test.sol

```
pragma solidity ^0.4.23;


contract Test {

    // 输入两个参数
    function add(uint8 arg1,uint8 arg2) public pure returns (uint8) {
        return arg1+arg2;
    }

}
```

SOLIDITY CONTRACT SOURCE CODE　CONTRACT BYTE CODE

SELECT CONTRACT TO DEPLOY

Test

编译正确

图 3-38　将 Remix 中编写好的合约代码复制到 Mist 中

Total Supply: 1 ACDCCAT
Holders: 1 addresses
Transfers: 1
Contract: 0x33378513b9f9406a43f6b0e7ebb
Social Profiles: Not Available, Update ?

Transfers　Holders　Inventory　Info　Read Contract　Write Contract　Comments

Sorry, we were unable to locate a matching Contract ABI or SourceCode for this contract.
If you are the contract owner, please Verify Your Contract Source Code here.

信息提示无法显示智能合约的代码

图 3-39　在区块链浏览器中无法显示智能合约代码

原因是合约的编译器版本问题导致“etherscan.io”不能根据 Mist 发布合约的“Bytecode”识别出这是一份符合 ERC20 标准的代币合约。出现这种情况，只能手动恢复，或者使用后续介绍的“Bytecode”方式在 Mist 中发布合约。

在 Mist 中填写好了“Solidity”代码之后，直接单击页面底部的“DEPLOY”按钮进行合约的部署，如图 3-40 所示。部署的本质就是发起一笔交易（Transaction）。等待以太坊矿工打包成功后，合约就部署成功了，并能在以太坊区块链浏览器上查询到。

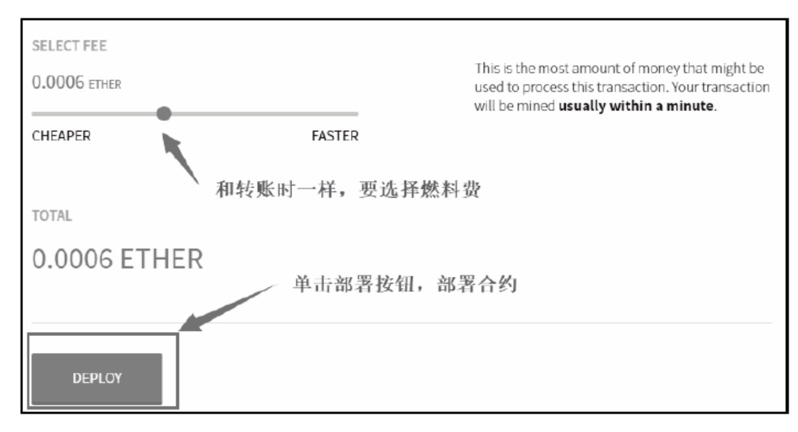



图 3-40 Mist 中部署合约也要选择燃料费

## 3.9.2 认识“ABI”

智能合约中的“Application Binary Interface”简称为“ABI”，中文全称是“应用程序二进制接口”。它的直观形式是一串“Json”字符串，“Json”里面包含下面的一些“Key”：

- name，字符串类型，对应的是当前项的名称。注意，根据 name 只能知道这个项的名称，究竟对于它是函数 function 还是一个 uint8 的变量并不清楚。
- type，字符串类型，标明当前的项是什么类型（是函数还是一个单纯的变量）。常见的 type 有下面的取值:
    - function 函数。
    - constructor 构造函数。
    - event 事件。
    - 变量类型，例如 address、uint256、bool 等。
- constant，布尔类型，代表当前项的操作结果是否会被写入到区块链上，是则为 true，反之为 false。
- stateMutability，字符串类型。stateMutability 拥有下面的取值:
    - pure，代表不会读和写区块链状态。
    - view，代表会读区块链状态，但不会改写区块链状态。
    - nonpayable，代表会改写区块链状态，例如转账 transfer 和授权 approve 这两个 ERC20 标准的函数就可以用来改写区块链。
- payable，布尔类型，代表当前的函数 function 是否可以接收 ETH 代币，可以则为 true，否则为 false，一般来说都是 false。
- inputs，其类型是 Json 数组，代表当前项入参的信息，内部会把每个参数的名称及其所对应的类型列出。一般来说，inputs 会跟随 type 是函数 function 或者事件 event 而含有值。inputs 中的 Json 变量除了 name 和 type 之外，还有下面两个变量:

   - Indexed，在 Solidity 代码的事件 event 中，其入参有设置为 Indexed 关键字，此时 Json 中的这个变量对应的值为 true，反之为 false。
   - components，该变量的类型是 Json 数组，当参数的 type 是 struct 结构类型时，该变量就会出现。
- outputs，和 inputs 的含义类似，其类型也是 Json 数组，代表的是当前项的返回值，内部表达是返回值的名称和类型。

inputs 和 outputs 如果没有值，便会默认地显示为[ ]。

- anonymous，布尔类型，它和“标准的事件（Event）”一节中介绍的“Indexed”的设置有关联，当为 true 的时候，在“event”中的入参即使是属于“Indexed”关键字的形式也不会保存到“Topic”中，反之则会。

下面是一个“ABI”的部分展示。根据以上对“ABI”各部分的认识可知，如果知道了一份智能合约“ABI”的 Json，那么也就能了解整个智能合约内部代码所实现的函数、事件、变量等信息。

同时，在以太坊源码中一般用“ABI”来做合约在代码层面的预初始化，在准备调用合约的函数时，可以先判断函数名称是否存在、入参类型是否匹配等操作。


```
[
    {
        "constant": true,
        "inputs": [],
        "name": "getHalfTotalSupply",
        "outputs": [
            {
                "name": "half",
                "type": "uint256"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "constant": true,
        "inputs": [],
        "name": "name",
        "outputs": [
            {
                "name": "",
                "type": "string"
            }
        ],
        "payable": false,
        "stateMutability": "view",
        "type": "function"
    },
    {
        "constant": false,
        "inputs": [
            {
```

```
                    "name": "_spender",
                    "type": "address"
                },
                {
                    "name": "_value",
                    "type": "uint256"
                }
            ],
            "name": "approve",
            "outputs": [
                {
                    "name": "success",
                    "type": "bool"
                }
            ],
            "payable": false,
            "stateMutability": "nonpayable",
            "type": "function"
        },
        ...
    ]
```

## 3.9.3　提取 ABI 和 Bytecode

在 Mist 智能合约发布界面的另一个输入框“CONTRACT BYTE CODE”中，并没有要求输入“ABI”，只需要输入“Bytecode”（字节码），但是在其他的一些智能合约发布工具软件中，需要用到合约“ABI”信息，包括在“geth”以太坊节点程序的控制台中进行合约发布的情况也需要“ABI”。

除了在合约发布的时候用到“ABI”之外，在代码层面的开发中也会用到，这部分内容将在后面的中继开发一章中讲到。

首先在 Remix 主页的右边工具栏上单击“Compile”，再单击“ABI”按钮进行 ABI 文本的复制，如图 3-41 所示。

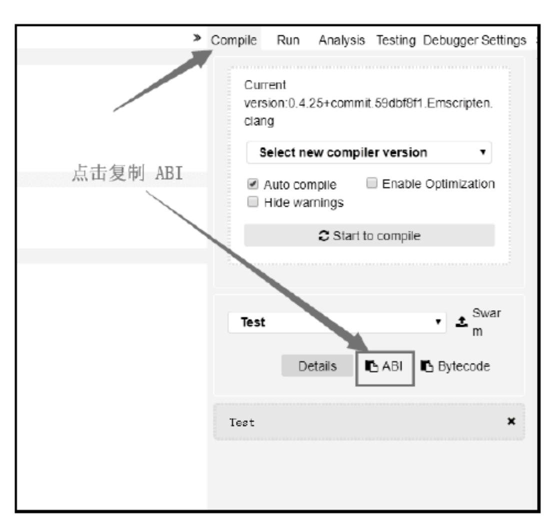



图 3-41　在 Mist 中复制出智能合约的 ABI 数据

接下来提取“Bytecode”。“Bytecode”的提取不能直接单击图 3-41 中的“Bytecode”按钮进行复制，应该单击“Details”按钮进入到详情页面，如图 3-42 所示。详情页面中会显示出当前编译成功的智能合约的所有信息。

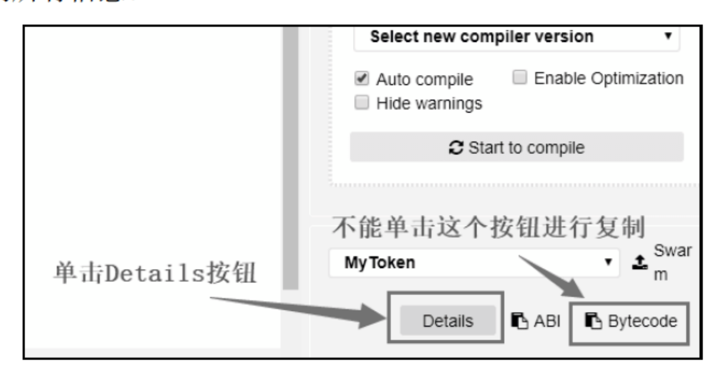



图 3-42　在 Mist 中复制出智能合约的 Bytecode 数据

单击“Details”按钮后，在弹出的页面中，滑动鼠标直至看到“WEB3DEPLOY”栏，找到里面实例中的“data”字段对应的内容，这就是我们要提取的“Bytecode”，它是一串完整的十六进制字符串，双击之进行复制即可，如图 3-43 所示。

```
WEB3DEPLOY

var mytokenContract = web3.eth.contract([{"constant":true,"inputs":[],"name":"getHalfTotalSupply","outputs":[{"name":"ha
var mytoken = mytokenContract.new(
   {
     from: web3.eth.accounts[0],
     data: '0x608060405260408051908101604052806001381526020017f4d79206669727374207468656e20636f696e000000000000000000
     gas: '4700000'
   }, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
         console.log('Contract mined! address: ' + contract.address + ' transactionHash: ' + contract.transactionHash);
    }
 })
```

复制 data 内容，它才是我们要取得 Bytecode 字段

图 3-43　Bytecode 字段的内容

## 3.9.4　使用 Bytecode 发布合约

推荐在 Mist 中使用“Bytecode”部署智能合约，这样能够在很大程度上避免由于 Remix 编译器版本和 Mist 编译器版本不同而导致的问题，即使发生了图 3-39 所提到的问题，也能方便地进行纠正，无须重新发布替换的合约，如图 3-44 所示。

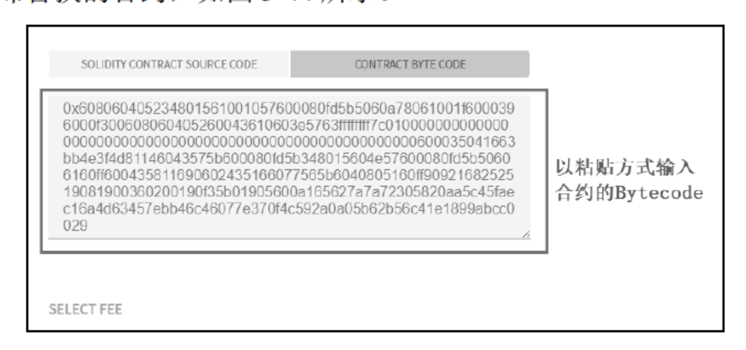



图 3-44　以粘贴方式输入合约的 Bytecode

填好了“Bytecode”后，直接单击“DEPLOY”按钮发起部署合约的交易。在发起交易的弹出页面中，可以看到交易中的“to”地址名称变成了“Create contract”， 如图 3-45 所示。其实“to”如果没有设置值（“to”地址不填的话），那么在我们发起交易的时候这个交易就是创建合约。这一点在以太坊的源码中已经做了声明。当然，如果我们在软件（例如一个钱包的转账页面）中不填写“to”地址，就会报错。这个报错是开发者自行设置的。

```
type txdata struct {
    AccountNonce uint64          `json:"nonce"    gencodec:"required"`   // 交易序列号
    Price        *big.Int        `json:"gasPrice" gencodec:"required"`   // gasPrice
    GasLimit     uint64          `json:"gas"      gencodec:"required"`   // gasLimit
    // to 交易的接收者地址，“nil means contract creation” 的意思是，空代表着创建智能合约
    Recipient    *common.Address `json:"to"       rlp:"nil"` // nil means contract creation
    Amount       *big.Int        `json:"value"    gencodec:"required"`   // 要交易的 token 数值
    Payload      []byte          `json:"input"    gencodec:"required"`   //  data 参数

    // Signature values
    // 下面的 v r s 就是签名的时候会赋值的，保存着签名后生成的数据
    V *big.Int `json:"v" gencodec:"required"`
    R *big.Int `json:"r" gencodec:"required"`
    S *big.Int `json:"s" gencodec:"required"`

    // This is only used when marshaling to JSON.
    Hash *common.Hash `json:"hash" rlp:"-"`
}
```

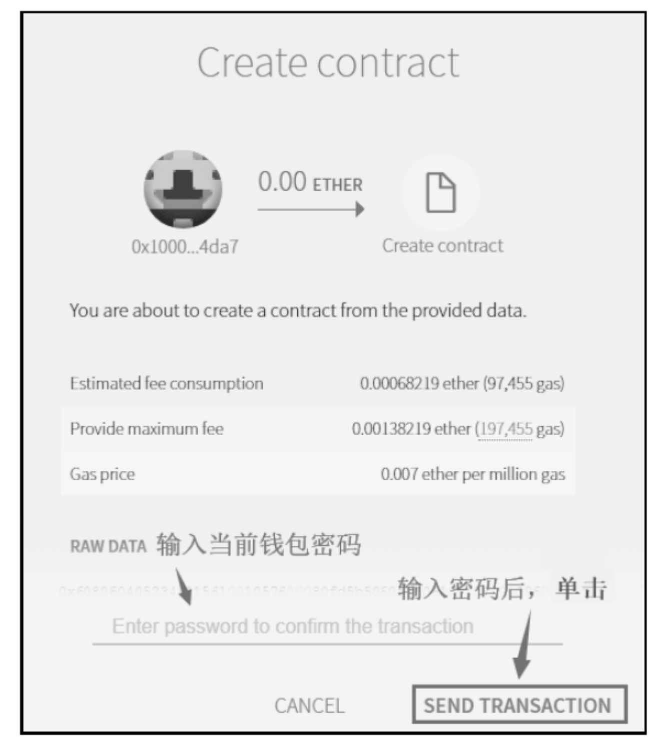



图 3-45　以太坊 Go 源码中对 to 地址的注释和 Mist 发起部署合约时候的界面

稍等一段时间，在“Latest Transactions”栏可以查看到合约的交易已经成功，如图 3-46 所示。

接下来，我们到区块链浏览器“etherscan.io”上进行查询验证。首先在 Mist 上面的交易详情页面复制哈希（hash）值，然后用浏览器打开“etherscan.io”进行搜索，如图 3-47 所示。

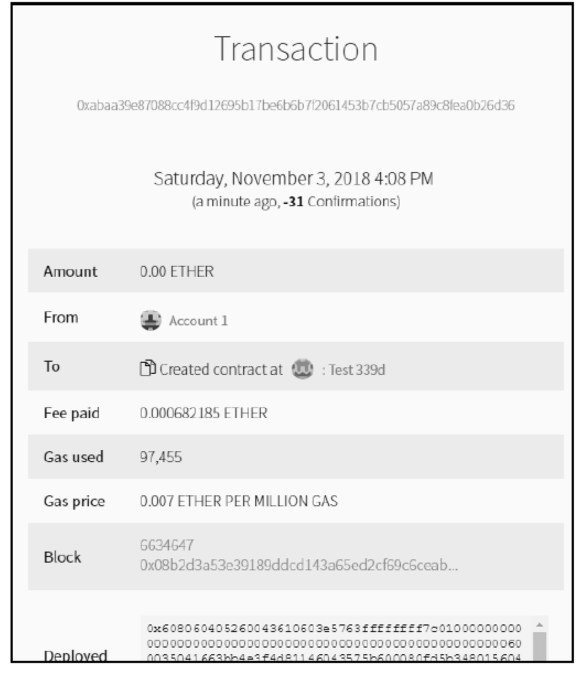



图 3-46　合约的交易已经成功

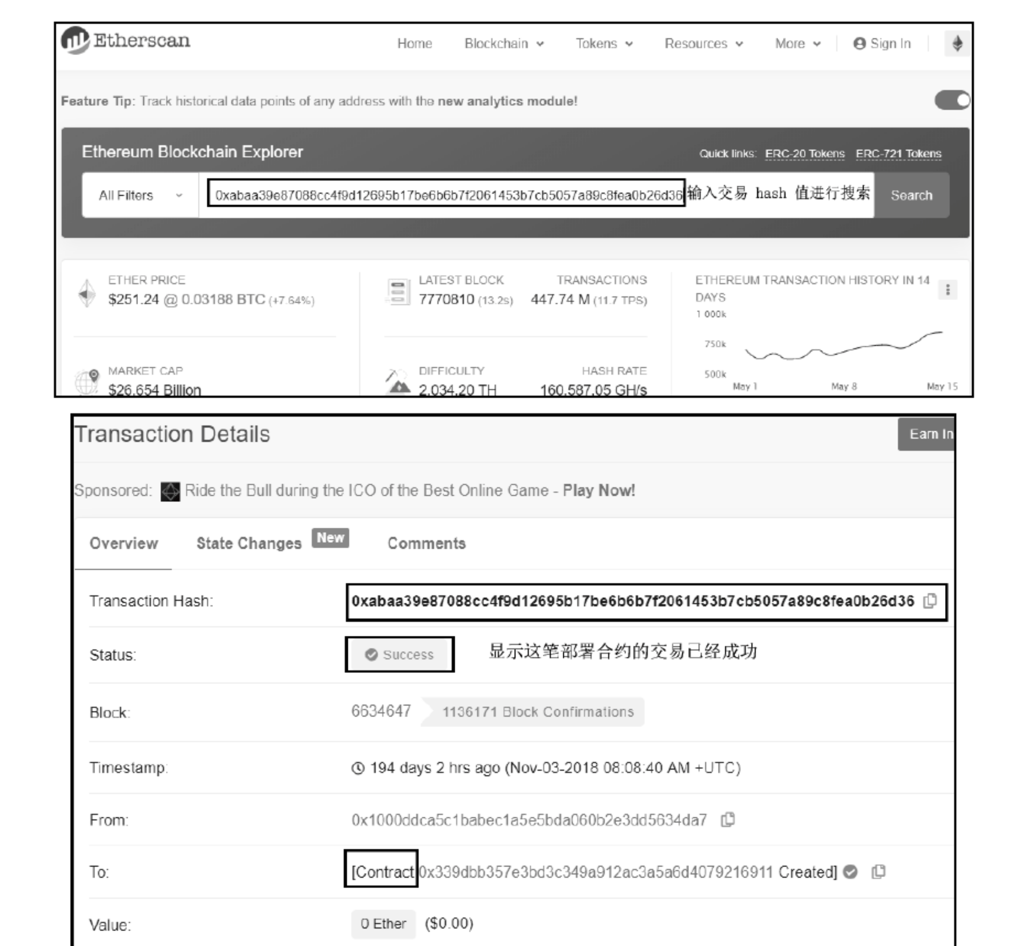



图 3-47　查看部署成功了的合约对应的交易信息

根据“etherscan.io”显示的结果，可以确认上面的合约已经成功发布到了以太坊的公链上。合约的以太坊地址是：

```
0x339dbB357E3BD3c349a912ac3a5A6D4079216911
```

该地址就是合约的唯一标识。如果要发布“ERC20”标准的代币智能合约，模仿上述的发布方法即可。

## 3.9.5 使用合约的函数

在上面的一节中，我们使用 Mist 发布了一个实现加法运算的智能合约，那么在智能合约发布之后，如何使用这份智能合约呢？

Mist 为我们提供了直接调用智能合约函数的页面，如图 3-48 所示。首先在 Mist 主页面上单击“CONTRACTS”按钮，进入到 Mist 已经发布或者添加了合约的列表页面，再找出我们想要调用的函数的合约，单击合约可进入到对应的详情页面。




图 3-48 在 Mist 中选择要使用的合约

在默认情况下，我们在合约列表页面看到的都是自己在 Mist 上成功发布了的合约。如果想调用别人发布的合约，怎么办呢？这种情况下需要手动添加，添加流程和添加“Custom Tokens”的方法大同小异。

首先在合约列表中单击“WATCH CONTRACT”按钮，在弹出的页面中按照图 3-49 的指示输入对应的信息，其中“JSON INTERFACE”就是我们前面谈到的“ABI”，如果要获取的“ABI”无法从合约编辑器（例如“Remix”）中获取，可以直接在区块链浏览器（例如“etherscan.io”）中查询获取，前提是必须知道要查询合约的以太坊地址。

例如，要添加的智能合约是代币“CDCC”的，知道它的以太坊地址是：

```
0x78021abd9b06f0456cb9db95a846c302c34f8b8d
```




图 3-49　在 Mist 中查看合约的信息

现在到“etherscan.io”中查询，在查询出来的页面中单击“Code”按钮，如图 3-50 所示。




图 3-50　在“etherscan.io”中查询合约

然后在对应的页面中滑动鼠标，找到“Contract ABI”一栏，复制“ABI”的内容，这就是我们要粘贴到“JSON INTERFACE”里面的内容，最后单击“OK”按钮即成功添加别人发布的合约，如图 3-51 所示。




图 3-51　找到 ABI 的内容复制

单击了想要调用的合约之后，在显示出的页面中就能看到当初在合约中所有定义好了的能够被外部调用的函数，因为我们要调用的是加法合约中的加法函数，所以会看到如图 3-52 所示的界面。尝试分别输入“Arg1”和“Arg2”，可以看到该合约的计算结果。






图 3-52　在 Mist 调用合约的函数

其他智能合约在 Mist 中的调用流程和本例相同，不再赘述。

截至目前，我们只是在界面化的层面直接调用了合约中的函数，实际开发中更多的是通过代码来进行智能合约相关函数的调用，相关内容我们将在第 4 章进行介绍。

# 3.10　小结

以太坊智能合约是目前以太坊 DApp 的主要实现形式，我们可以通过发布一份智能合约来发布一个 DApp 应用。本章从智能合约使用的完整流程开始，介绍了如何使用 Remix 编写智能合约以及使用 Mist 发布智能合约。

需要注意的是，在编写合约方面，并没有从合约的 Solidity 编程语言方面进行讲解（有关 Solidity 语言的内容，读者可参考专门的资料进行学习），而是使用了两个例子进行学习。实现加法功能的智能合约是其中一个最为简单的入门级例子，通过该例可以帮助我们认识智能合约是一个使用代码编写的程序的本质。在加法例子之后结合工具软件，对经常被用于以太坊上发币所使用的 ERC20 标准代币合约的实际应用进行了详细介绍，包括在 Remix 编辑器中进行代币合约的 Solidity 代码编写，以及得到 ERC20 代币合约的“Bytecode”后再使用 Mist 进行发币等内容。

# 第 4 章

# 实现以太坊中继——基础接口

在前面各章中，我们主要介绍的是以太坊 DApp 开发的相关基础知识，从本章开始，我们将通过一个 DApp 开发实例——以太坊中继，对这些知识进行综合应用，以帮助读者掌握如何自己动手开发 DApp 项目的实用技能。

本章首先介绍以太坊中继的基础接口及相关概念，在下一章我们将会深入讲解以太坊中继的应用开发。

## 4.1 认识以太坊中继

首先，我们来认识一下以太坊中继。通过如图 4-1 我们可以看出以太坊中继器（以下简称为中继）在基于服务架构中的位置。

以太坊中继在服务集群中充当的是一座连接传统服务器端和以太坊区块链的桥梁，也可以看作是服务分离的一部分。中继负责公链上相关功能的实现，几乎囊括了目前以太坊 DApp 的绝大部分功能。

以太坊中继能够直接提供但不限于下面的功能：

- 接受其他服务端链上的相关服务请求，然后查询被请求的链节点，获取对应的数据后再返回给请求的服务者，例如交易记录的查询、代币余额查询等。
- 发起交易的功能，包括以太坊 ETH 转账和 ERC20 代币转账。这项功能经常出现在目前交易所开发的应用中，一般对应用户的提币操作，用户在移动端发起提币请求，传统服务器接收请求后，在内网中访问以太坊中继，把交易信息发给中继，然后中继从交易所的对公账户中转出对应数量的币值。
- 用户以太坊钱包的创建。这个功能一般对应于中心化交易所帮助用户在服务器端创建钱包的功能，也是目前常见的功能。

- 对链上区块相关事件的监听。该功能特别重要，其中也包含了我们在智能合约一节中所谈到的交易结果是否被成功监听。目前能够监听的事件包括但不限于下面的各项：
  - ERC20 代币授权“Approve”事件。
  - 代币转账“Transfer”的结果事件。
  - WETH 代币置换 ETH 事件。
  - ETH 置换 WETH 代币事件。
  - 新区块生成事件。
  - 遍历完一个区块事件。
  - 区块链分叉事件。

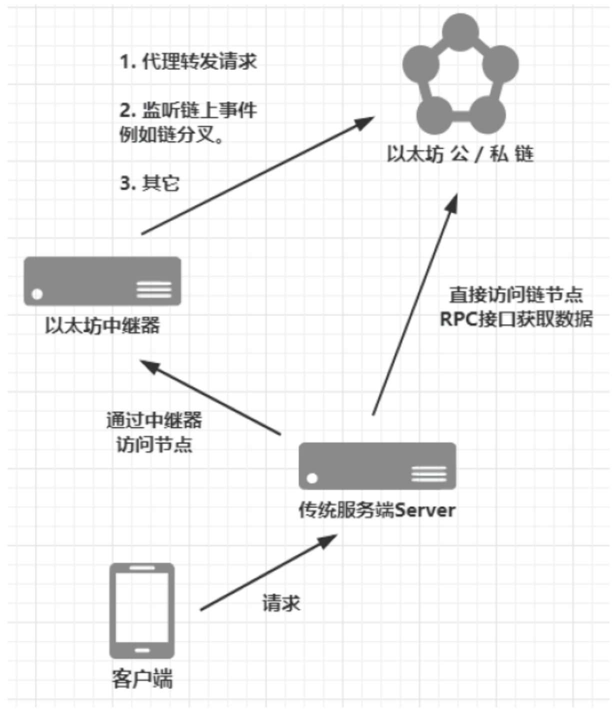



图 4-1　以太坊中继

以上是常见的以太坊事件，具体的还有很多，例如在“标准的事件（Event）”一节中提到的——使用 Solidity 编写智能合约的时候可以定义自己想要的“Event”事件，也就是说，以太坊中继还可以监听“自定义事件”等。

# 4.2　区块遍历

在中继可实现的功能中，事件的监听是开发难度最高的，因为其他的以太坊接口调用如果不考虑服务分离，可以并入到传统服务器端的功能模块中去。

事件监听的技术原理主要是通过获取一个区块内部的交易信息并解析交易信息内的“Event Log”（事件日志）来达到目的。在以太坊目前提供的 Web3.js 库中，有区块事件的监听函数，但

Web3.js 一般用于客户端，且存在以下不足：

（1）如果客户端的进程被杀死，监听动作就会丢失。

（2）若没对上次最后遍历成功的区块号进行存储，重新启动的时候将会造成时间段内新生成区块的数据丢失。

（3）完整的监听流程比较消耗客户端的设备资源，影响用户体验。

因此，区块的遍历及其内部数据获取后的存储应该在后端服务中进行，也就是以太坊中继器应该包含该功能。

为什么一定要有监听区块事件的功能？下面我们通过一个在交易所中实际用到的例子来进行阐述。

中心化交易所基本都具备用户提币（提现）的功能。这里的提币指的是把币从交易所转到用户公链上的钱包地址中去，毫无疑问，这是一个涉及在公链中发送交易的操作。我们知道，以太坊的交易不会马上知道刚发送的交易是否成功或失败，只能够知道一笔交易的哈希值，但是在用户发起提币请求后，人工审核发起转账，肯定要在某一个时间点通知客户端或者数据库有关转账请求的状态更新，例如把提币请求更新为提币成功。

基于以上的例子，如果在客户端对交易返回的哈希值进行不断地监听——监听交易在什么时候成功，这将会存在上面谈到的在客户端进行交易事件监听的 3 个问题，所以是不可取的。

这个时候如果让中继不断地遍历每一个区块，按照区块高度来逐个遍历，当提币的交易最终在链上成功了，它就会被打包进一个区块中。自然地，在我们遍历到这个区块时，就能把里面的所有交易信息提取出来，当发现了对应的哈希存在时，证明提币到账了。除此之外，还能把从每个区块中遍历出的交易记录保存到数据库中，并做一定的字段过滤。例如，只保存一种 ERC20 代币的转账记录，有了这些记录，能让客户端发起交易查询时直接查询以太坊中继来得到结果，而不是通过以太坊节点的接口查询。

上面就是基于以太坊交易的例子（在交易所提币时），是以太坊中继监听事件的体现之一。此外，还有分叉事件的监听处理，即当监听到了分叉事件时有可能会对某些数据更新进行回滚操作等。

图 4-2 是一个区块遍历的大致模型图。

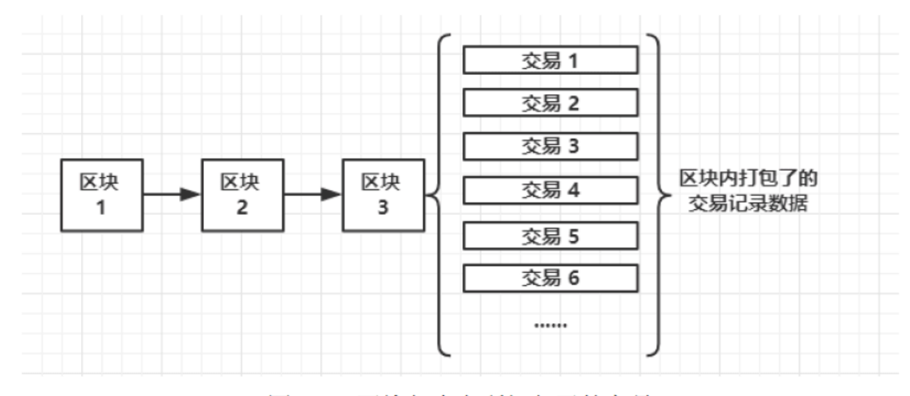



图 4-2　区块包含它所打包了的交易

在以太坊的机制中，链上的授权、交易、合约发布等事件都是一条条的交易信息，我们把每笔交易信息提取出来再进行分类应用，就能够实现不同的功能。

# 4.3 RPC 接口

无论 C/S 还是 B/S 技术架构，客户端和服务器端的交互都是通过请求与响应的方式进行的，如图 4-3 所示。

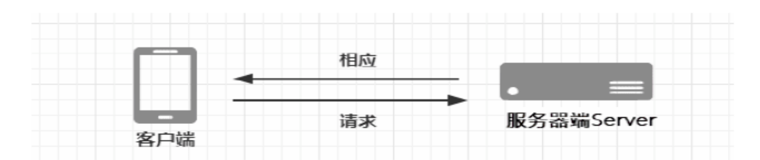



图 4-3 一般的客户端访问服务器端的请求与响应

最为开发者熟悉的客户端请求服务器端的接口就是“RESTful API”。这类“API”的特点是，客户端可以通过使用“GET”或“POST”的方式进行请求。目前，服务器端接口除了“RESTful API”这种类型，还有一种就是我们多次提到的“RPC”接口类型。

RPC（Remote Process Call，远程过程调用）和“RESTful API”一样，能够被客户端应用于与服务器端的交互，这是两种接口最大的共同点。下面我们从协议及实现的角度来认识一下这两种接口的不同。

图 4-4 所示是我们熟知的 OSI 七层网络通信模型。

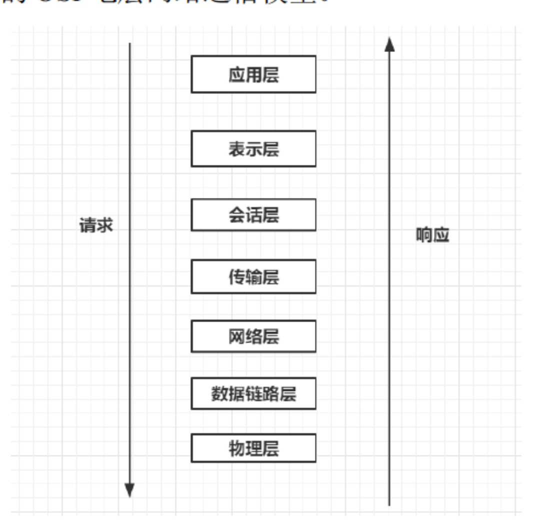



图 4-4 OSI 七层网络通信模型

从协议角度来看，“RESTful API”接口在应用层基于的协议就是 HTTP 或 HTTPS，在经过了应用层到达传输层时就会使用 TCP 或 UDP 协议。“RESTful API”为开发人员提供了多种请求方式，GET/POST 请求方式只是其中的两种，还有 Put、Delete、Head、Option 4 种请求方式。由于 HTTP/HTTPS 协议已经被开发得很完善了，因此开发人员在编写“RESTful API”接口程序时可以大幅地减少开发时间。

“RPC”接口在基于通信协议方面的实现有多种，主要有下面的两种：

（1）同“RESTful API”一样，基于应用层 HTTP/HTTPS 协议的实现。

（2）基于传输层的 TCP 协议的实现，也被称为 Socket（套接字）的实现。

从协议的实现角度来看，请求和接收响应在应用层发出和在传输层发出有很大区别。如果是基于 HTTP/HTTPS 协议实现，在速度方面，“RPC”接口和“RESTful API”接口几乎无差别，但基于传输层的 TCP 协议实现“RPC”接口，“RPC”接口除了因为在数据传输流经的层级上比“RESTful API”少而整体比它快之外，传输时的整体数据报层面还少了 HTTP/HTTPS 的头部数据量及组装的时间损耗。也就是说，在实现同样功能的情况下，“RPC”不仅请求与响应速度要比“RESTful API”快，且数据量也相对要少。

此外，因为使用“TCP 协议”实现的“RPC”接口不像应用层的 HTTP/HTTPS 协议那样，已经为我们做好了很多复杂的事情，包含数据的编码、解码等，在传输层上，如果我们不依赖第三方框架来自己动手实现一套“RPC”框架，那么从请求到响应及其解码数据的过程，其难度都是比较大的。然而，由于现今的计算机编程语言都提供了很多成熟的“RPC”框架，实际上开发者也可以简单地实现“RPC”类型的接口。

从数据传输格式上进行分类，常见的“RPC”框架有以下几种：

- JSON-RPC
- XML-RPC
- Protobuf-RPC
- SOAP-RPC

所谓的“RPC”协议，就是规范了一种客户端和实现了“RPC”接口的服务器端交互时的数据格式。

“RPC”接口实现的大致流程是：服务的调用方按照规范好了的编码方式把某个“RPC”接口的函数名称和参数进行序列化编码后，发送到服务的提供方，即服务器端，服务器端再通过反序列化后把对应的参数提取出来，然后通过调用相关函数，最后把结果返回给服务的调用方，完成整个流程。

# 4.4　以太坊接口

目前以太坊 Go 语言版本的节点源码中所有对外提供服务的接口都是“RPC”类型的，源码地址是 https://github.com/ethereum/go-ethereum。

为了方便开发者学习使用，以太坊的官方开发团队采用 JavaScript 语言开发了一整套以太坊节点“RPC”接口的开源库，名称为“web3.js”库，官方开源地址是 https://github.com/ethereum/web3.js。

除了官方基于 JavaScript 的版本之外，我们到“GitHub”上进行搜索可以发现，截至到目前，“web3.js”库已经拓展出了其他语言的版本，这些不同语言版本的 web3.js 库方便了以太坊相关应用的开发，如图 4-5 所示。

以太坊源码提供的“RPC”接口有很多，除了获取余额的接口“getBalance”和交易的接口“sendRawTransaction”之外，还有估算一笔交易的燃料费接口“estimateGas”等。当我们想要使用某个接口的时候，可以查看官方的接口文档，文档列举了所有“RPC”接口的信息、包含请求的参数以及返回结果的结构等。

官方例子“web3.js”文档链接是 https://web3js.readthedocs.io/en/1.0/。

官方“RPC”接口的完整文档链接是 https://ethereum.gitbooks.io/frontier-guide/。

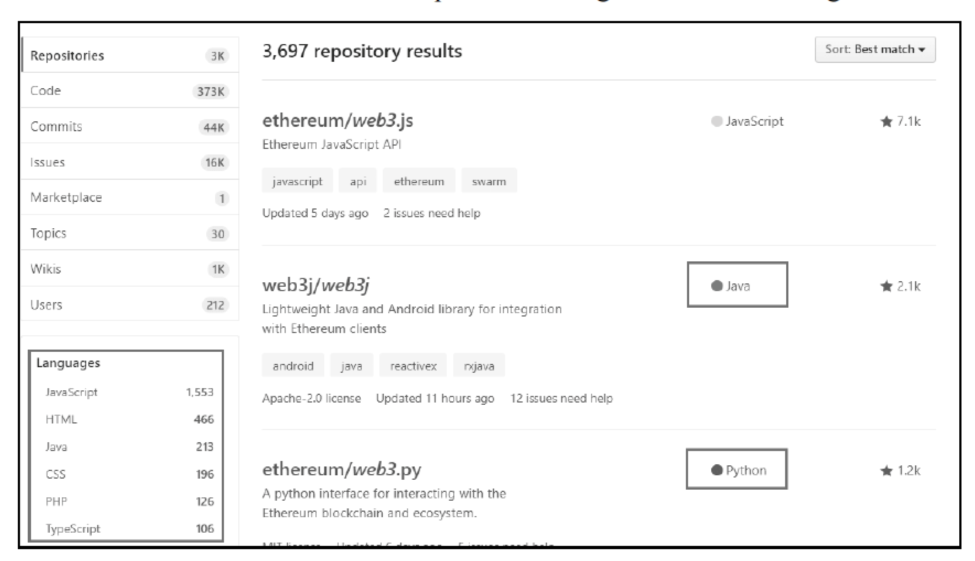



图 4-5 web3.js 在 GitHub 上的不同语言版本

## 4.4.1 重要接口详解

根据 RPC 接口文档，我们在开发以太坊中继时主要用到的“RPC”接口是“eth”部分，这部分的接口涵盖了区块和交易这两大模块，如图 4-6 所示。

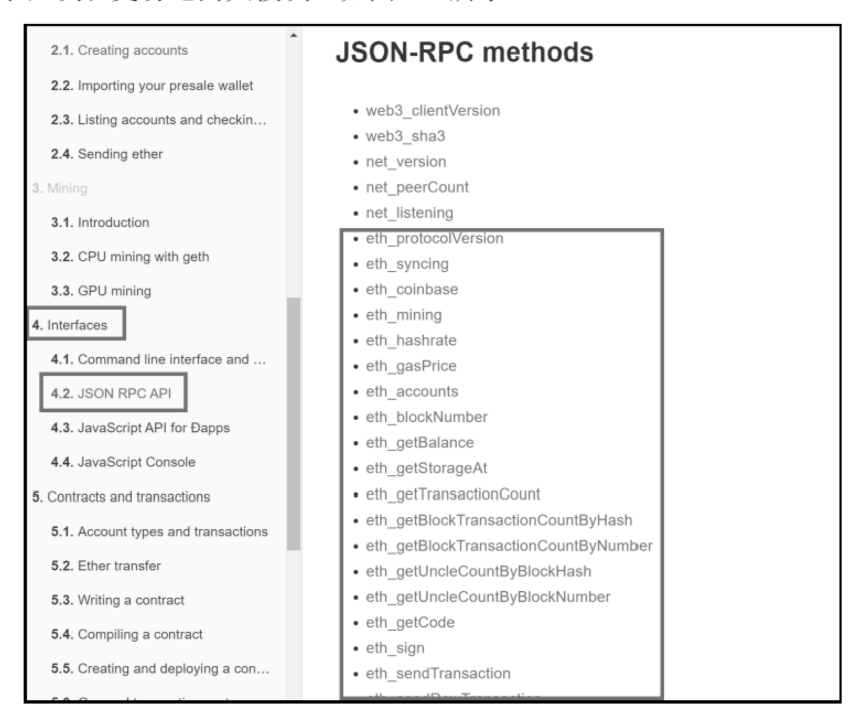



图 4-6 “RPC”接口完整文档的接口函数（方法）列表

下面我们介绍的几个 RPC 接口在开发以太坊中继时都会用到，请务必了解每一个接口的作用，

特别是接口的函数名称和功能。在讲解接口之前，我们先介绍 3 个重要的参数：

- genesis，代表的是最早的，早期的源码版本中等同于 earliest。
- pending，代表的是等待或挂起状态中的。
- latest，代表的是最新完成的。

这 3 个参数都会出现在每一个接口中，关于它们的实际使用，我们会在下面介绍具体的接口时进行详细说明。下面我们介绍几个重要的接口。

### 1．eth_blockNumber

该接口可以根据传参获取 3 种类型的区块高度，也就是区块号。其参数包括 latest、pending、和 genesis。其中，“latest”参数获取的是当前链上最新生成的区块的高度；“pending”获取的是当前正在被矿工开采的区块，代表正在打包交易的区块，一个区块在打包完一定量的交易后才会完整上链；“genesis”代表的是创始区块的区块号，也就是第一个区块的高度，要注意的是，最早的区块号不一定就是 0，因为在生成的时候是可以指定不为 0 的。

### 2．eth_getBlockByNumber

该接口根据区块高度获取区块的部分信息，是在遍历区块时主要用来获取区块数据信息的一个接口。它能够提供下面的数据：

（1）区块头部的部分字段。
（2）所有打包在这个区块中的交易的哈希数组。

Block 区块的结构体的定义如图 4-7 所示。

```
27  type Block struct {
28      Number           types.Big    `json:"number"`
29      Hash             common.Hash  `json:"hash"`
30      ParentHash       common.Hash  `json:"parentHash"`
31      Nonce            string       `json:"nonce"`
32      Sha3Uncles       string       `json:"sha3Uncles"`
33      LogsBloom        string       `json:"logsBloom"`
34      TransactionsRoot string       `json:"transactionsRoot"`
35      ReceiptsRoot     string       `json:"stateRoot"`
36      Miner            string       `json:"miner"`
37      Difficulty       types.Big    `json:"difficulty"`
38      TotalDifficulty  types.Big    `json:"totalDifficulty"`
39      ExtraData        string       `json:"extraData"`
40      Size             types.Big    `json:"size"`
41      GasLimit         types.Big    `json:"gasLimit"`
42      GasUsed          types.Big    `json:"gasUsed"`
43      Timestamp        types.Big    `json:"timestamp"`
44      Uncles           []string     `json:"uncles"`
45      Transactions     []string     // 所有被打包进来的交易的 hash 数组
46  }
```

图 4-7　以太坊 Go 版本源码中 Block 区块结构体的定义

### 3．eth_getTransactionByHash

该接口可根据一笔交易记录的哈希值获取这笔交易的详细信息。该接口提供了交易查询功能，在获取区块信息后，从区块信息中获取被打包的交易的哈希值，进行全部交易记录信息的提取。处于 pending（等待或挂起）状态的交易，返回的将会是空，我们也可以根据接口的这个特点来判断一笔交易是否成功，它能够给我们提供如图 4-8 所示的数据。

```
67  type Transaction struct {
68      Hash             string    `json:"hash"`
69      Nonce            types.Big `json:"nonce"`
70      BlockHash        string    `json:"blockHash"`
71      BlockNumber      types.Big `json:"blockNumber"`
72      TransactionIndex types.Big `json:"transactionIndex"`
73      From             string    `json:"from"`
74      To               string    `json:"to"`
75      Value            types.Big `json:"value"`
76      GasPrice         types.Big `json:"gasPrice"`
77      Gas              types.Big `json:"gas"`
78      Input            string    `json:"input"`
79      R                string    `json:"r"`
80      S                string    `json:"s"`
81      V                string    `json:"v"`
82  }
83
```

图 4-8　以太坊 Go 版本源码中 Transaction 交易结构体的定义

图 4-8 交易数据中各个字段的含义是：

- Hash，当前交易的哈希值。
- Nonce，当前交易对应的系列号。
- BlockHash，当前交易被打包进区块的哈希值。
- BlockNumber，当前交易被打包进区块的高度，也就是区块号。
- TransactionIndex，当前交易在区块的所有打包了的交易数组中的下标。
- From，发起交易的以太坊地址。
- To，这笔交易发往的以太坊地址。注意，这里不能直接看作是收款人的以太坊地址。在合约类代币交易中，就是智能合约的以太坊地址。
- Value，这笔交易的交易额，对应的是以太坊 ETH 的数量。如果是合约类代币交易，这个数值应该是 0。
- GasPrice，这笔交易每笔燃料（Gas）的价格，详情参考“交易参数的说明”一节。
- Gas，对应的就是“GasLimit”，详情参考“交易参数的说明”一节。
- Input，就是“交易参数的说明”一节中所讲的“data”。
- R、S、V，和交易签名相关的 3 个字段，是验签时所需要的字段。

### 4．eth_getTransactionReceipt

该接口可根据一个交易的哈希值来获取这笔交易收据的详情。注意，该接口返回的数据在一定程度上和 eth_getTransactionByHash 是相同的，但该接口返回的数据更详细，例如交易的日志“Logs”，这也是我们进行事件监听最主要的数据源。和 eth_getTransactionByHash 一样，处于 pending 状态的交易，返回的将会是空。该接口能提供如图 4-9 所示的数据。

```
137 type TransactionReceipt struct {
138     BlockHash         string     `json:"blockHash"`
139     BlockNumber       types.Big  `json:"blockNumber"`
140     ContractAddress   string     `json:"contractAddress"`
141     CumulativeGasUsed types.Big  `json:"cumulativeGasUsed"`
142     From              string     `json:"from"`
143     GasUsed           types.Big  `json:"gasUsed"`
144     Logs              []Log      `json:"logs"`
145     LogsBloom         string     `json:"logsBloom"`
146     Root              string     `json:"root"`
147     Status            *types.Big `json:"status"`
148     To                string     `json:"to"`
149     TransactionHash   string     `json:"transactionHash"`
150     TransactionIndex  types.Big  `json:"transactionIndex"`
151 }
```

图 4-9　以太坊 Go 版本源码中 TransactionReceipt 收据结构体的定义

除去与 eth_getTransactionByHash 相同的数据字段外，该接口其他数据字段的含义如下：

- CumulativeGasUsed，该字段和 GasUsed 不一样，目前还没有一个通俗的解释，包含官方文档对它的描述也不清晰。它所代表的是当前交易所在区块的交易列表中，当前 TransactionIndex 之上包含自身的其他所有交易的 GasUsed 之和。例如，TransactionIndex=5，那么它的 CumulativeGasUsed 数值为下标是 0、1、2、3、4、5 的交易的 GasUsed 数值之和。
- ContractAddress，合约地址，这个字段只有在当前交易是合约创建的情况才会有值，对应的是新创建合约的以太坊地址，其他情况都是空。
- GasUsed，实际消耗的燃料费，它和 GasLimit 的关系请参考“交易参数的说明”一节。
- Logs，由当前交易生成的事件（Event）日志。
- Root，当前交易在默克尔树根节点的哈希值。
- Status，如果这笔交易最终是成功的，那么其值为 true，也就是 1，否则为 false。

### 5. eth_getBalance

这个接口获取的是某个以太坊地址的 ETH 数值，即 ETH 余额。注意，它不是获取 ERC20 代币及其他代币的余额。

### 6. eth_sendTransaction 和 eth_sendRawTransaction

这两个接口在“以太坊交易”一节中已经做过介绍，是所有交易的触发接口。其中，eth_sendRawTransaction 可以完成的交易类型包含 eth_sendTransaction 的类型。

### 7. eth_getTransactionCount

该接口可根据一个以太坊钱包地址获取基于当前钱包地址的交易序列号 Nonce。该接口传入的第二个参数就是“genesis”“pending”和“latest”，各参数分别对应下面的作用：

（1）取 genesis 时，获取当前以太坊地址第一次发起交易时的 Nonce 序列号。

（2）取 pending 时，获取当前以太坊地址提交了的且正处于 pending 状态等待被区块打包的交易订单所对应的 Nonce 序列号。请注意，如果当前查询的地址没有处于 pending 状态的交易，那么它将返回与 latest 一样的 Nonce 号。

（3）取 latest 时，获取当前以太坊地址提交了且被区块成功打包了的交易订单所对应的 Nonce 序列号加一的值。举个例子，地址 A 最后一笔成功交易对应的 Nonce 为 4，当调用接口传入该参

数的时候，获取的结果就是 5。

（4）在 eth_getTransactionCount 中，Nonce 查询满足：pending ≥ latest ≥ genesis。

#### 8. eth_getCode

该接口根据以太坊地址判断当前地址是非合约地址还是合约地址。如果是非合约地址，那么返回值是“0x”；如果是合约地址，那么返回值则是智能合约当初创建时的十六进制码。

#### 9. eth_estimateGas

该接口用来估算一笔交易要消耗多少 GasLimit，注意所估算出的值没有涉及 GasPrice。入参中的“data”是被估算的数据量，估算的结果数值和数据量成正比。这个接口在钱包发起交易时让用户选择燃料费为多少的功能上会用到，体现在燃料费进度条中的最大值上。最大值一般就是这个接口返回的数值。

#### 10. eth_call

这是以太坊用来访问智能合约函数的万能“RPC”接口，意思是任何智能合约上的公有函数都能使用这个接口进行访问。前面提到，“eth_getBalance”是用来查询以太坊 ETH 余额的，如果要查询非 ETH 的代币余额，例如 ERC20 代币的余额，就可以通过 eth_call 来进行查询。这个接口的入参和“sendTransaction / sendRawTransaction”接口是一样的。

在 eth_call 中，我们可以通过控制 data 参数值中的“methodId”及所调用的合约的入参来达到访问不同智能合约函数的目的。

此外，请务必了解，eth_call 接口是只读的，它不会造成区块链上数据的改变，自然也就不会造成真实数据的改变。以 ERC20 代币转账为例，一般转账用 sendRawTransaction，是否可以用 eth_call 来实现转账呢？因为 eth_call 接口的定义是，可以访问智能合约中的所有函数，既然 transfer 是智能合约中的一个函数，那么 eth_call 就能访问该函数，而且 sendRawTransaction 转账到 EVM 虚拟机执行 data 时也会用到合约中的 transfer 函数，所以是否可以用 eth_call 实现转账，答案是否定的。

这里我们对 eth_call 只读特性从源码层面做一下说明。

以太坊虚拟机 EVM 在执行合约代码时会先使用区块号 blockNumber 找出对应的区块信息，再系统地根据这个区块的数据实例化一个名为 status 的变量，由 status 来实例化对应的虚拟机 EVM 实例，然后根据区块提供的信息来实例化合约代码中的变量值，即智能合约的函数被执行前它的变量值是由当前的区块高度来决定的。如此一来，eth_call 在调用 transfer 函数时会直接在代码的内存层面进行值的修改，而并没有广播出去和修改到数据库层面。

下面我们继续借助一个例子中进行进一步的直观认识。

请看下面的例子：

block 为 1000 时，余额值为 10，然后使用 eth_call 在 1000 高度转账了 1 个代币，eth_call 会根据 1000 高度的状态实例化 EVM，余额为 10；待 EVM 执行 eth_call 的 data 后，此时在内存层面余额变为 9，当前的 EVM 实例在执行完 eth_call 后释放，不会被广播；等调用 eth_call 取 balanceOf 时，eth_call 会再实例化一个 1000 高度状态下的 EVM，余额为 10。此时再用 sendRawTransaction 转账 1 个代币，这笔交易被广播，假设被打包进了 1001 区块高度。此时调用 eth_call，将实例化出来一个 1001 高度状态下的 EVM，balanceOf 读取余额为 9。

## 4.4.2　节点链接

要想访问以太坊节点的接口，需要知道接口的链接。这和常规的服务器端开发，获取服务器端的“IP”和端口“Port”是一样的。目前获取节点链接的方案主要有以下两种：

- 自行购买服务器，然后启动以太坊节点服务程序，例如“Geth”，再获取节点程序提供的接口链接。
- 使用第三方服务平台提供的节点链接。

对于第一种方法，操作流程是先下载对应的以太坊节点版本源码，然后根据文档的编译方法进行自编译，生成可执行程序，再部署到服务器端。需要注意的是，以太坊节点的源码不仅仅有版本号不同的版本，还有不同计算机语言的版本，官方的“Geth”是“Golang”语言开发的版本，源码链接是 https://github.com/ethereum/go-ethereum。

第一种方案整体实施起来有以下优缺点：

（1）可自定义性高。

- 可以自行配置节点启动的配置文件。
- 可自行修改源码进行二次开发后再编译。
- 可自行设置服务网关、节点集群等相关的运行方式。
- 可参与公链挖矿，赚取以太坊 ETH 的收入。

（2）技术要求难度相对来说比较高。

- 要求使用者必须掌握和了解配置文件中各项属性的含义及其影响。
- 要求使用者必须懂一定的服务器端运维的知识。
- 要求使用者懂得对应节点编写语言的编译命令，甚至使用过该门语言。

（3）自运维成本高。

- 要求监控节点服务器的运行情况。
- 要求实现节点程序，因为某些问题被杀死后能够进行自重启。
- 在集群情况下，要求保证各服务的稳定性。

（4）需要自付费服务器等产品的花销。

第二种方案在学习阶段以及线上业务不需要高度自定义化的情况下使用。相比第一种方案而言，第二种方案的优缺点刚好和第一种方案相反。除此之外，第二种方案也是最方便的，因为只需要一个链接即可。

以下的所有内容我们主要基于第二种方案进行讲解，以及如何获取免费的节点链接。

## 4.4.3　获取链接

我们可以从网站 https://infura.io/来免费获取节点链接。Infura 是国外一个托管以太坊节点的集

群，从国内访问暂时不用翻墙，且还允许开发者免费申请属于自己的以太坊节点链接（包含以太坊主网和测试网络），可以说十分方便和齐全！

打开 Infura 的主页后，单击“GET START FOR FREE”按钮就能进行链接的申请，如图 4-10 所示。

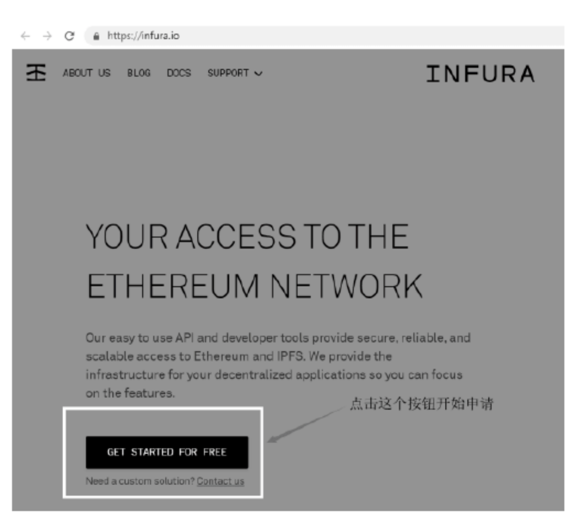



图 4-10　Infura 的创建账号入口按钮

单击进行申请后，首先完成账号的注册，如图 4-11 所示。

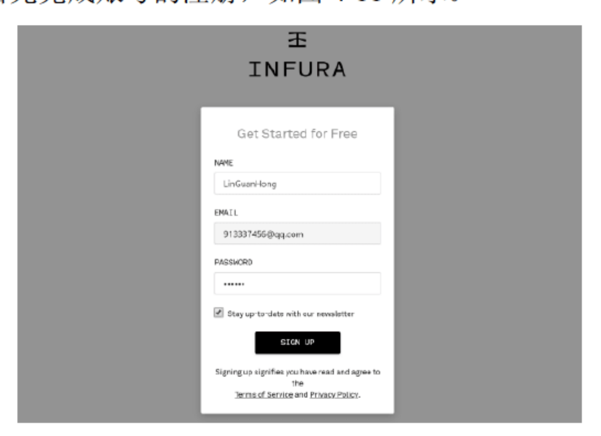



图 4-11　Infura 的账号注册页面

填写好资料后，单击“SIGN UP”，此时 Infura 会发送一封验证邮件到你的邮箱，登录你的邮箱查看这封邮件，在邮件内单击“CONFIRM EMAIL ADDRESS”链接，进行外部链接的访问，即可完成注册，如图 4-12 所示。

待成功进入控制台页面，需要先创建一个项目，以方便管理。根据提示，我们创建一个名为“test”的项目，如图 4-13 所示。

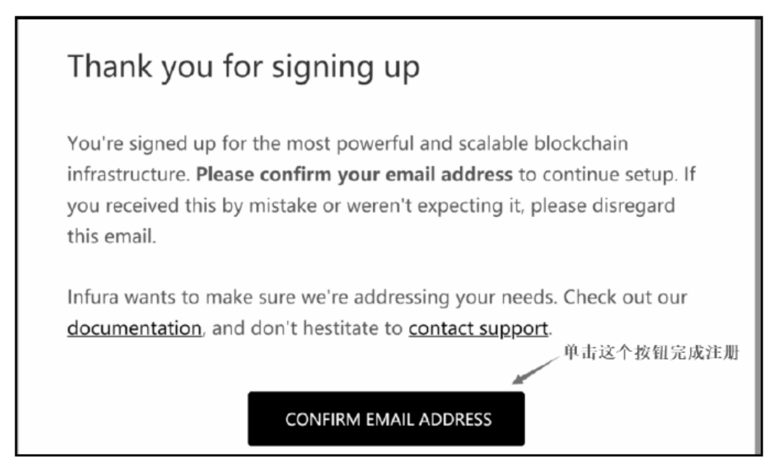



图 4-12　Infura 注册账号时发送的验证邮件




图 4-13　创建好账号后创建新项目

输入项目名称后的效果如图 4-14 所示。

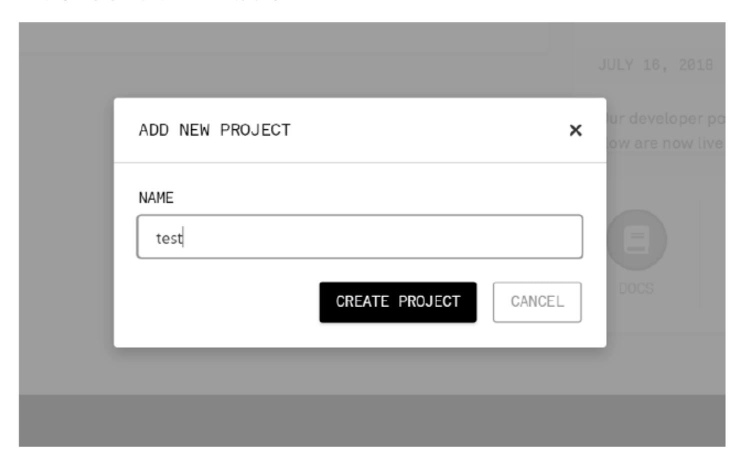



图 4-14　设置项目的名称

输入名称后按下回车键，创建成功后，就可以在控制台页面看到分配给我们的以太坊节点网络链接。把鼠标移动到“ENDPOINT”下的链接中，可以看到完整的节点链接，其中按钮“MAINNET”这个节点链接的是主网，“main”就是主要的意思，同时用鼠标左键单击这个按钮，就能看到下拉列表的网络类型选择，每种网络类型对应有节点链接，如图 4-15 所示。

主网：https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b。

“ROPSTEN”测试网络：https://ropsten.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b。

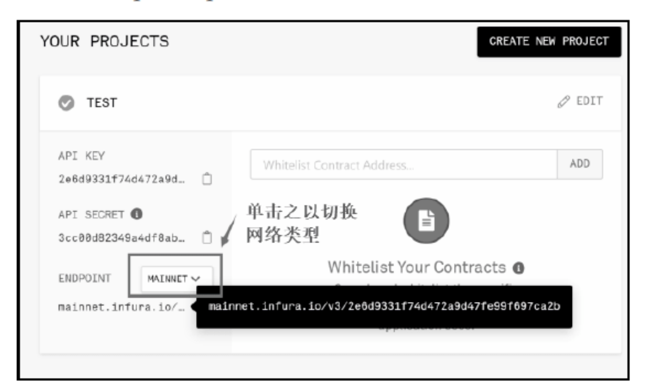



图 4-15 创建好项目后获取不同类型的节点链接

节点的分类（见图 4-16）：

（1）MAINNET，代表的是以太坊主网，如果我们使用这个链接来测试，那么所有的资产转换都是真实的资产。对应的区块链浏览器链接是 https://etherscan.io/。

（2）ROPSTEN，代表的是以太坊的测试网络，使用的共识算法是“PoW”，意味着可以采用挖矿来获取测试的 ETH。如果使用这个链接来测试，那么资产都是测试的，这些测试的资产可以通过挖矿或申请获得。对应的区块链浏览器链接是 https://ropsten.etherscan.io/。

（3）KOVAN 和 PINKEBY 一样，也是以太坊的测试网络，但是使用的共识算法是“PoA”（Proof-of-Authority，权威证明），“PoA”是由若干个权威节点来生成区块，其他节点则无权生成，自然地也就不能使用挖矿的形式获取 ETH 测试币，只能够通过申请获得。它们对应的区块链浏览器链接分别是：

https://kovan.etherscan.io/

https://rinkeby.etherscan.io/




图 4-16 获取不同类型的节点

## 4.4.4 进行测试

在上面的一节中，我们获取了以太坊的节点链接，现在我们来进行一下简单的测试，看看是

否可以真的对节点进行访问。

首先选择使用主网的链接进行简单的测试，把链接从 Infura 的管理台页面复制下来，在链接前面加上“https”。这点不要漏了，目前 Infura 提供的节点链接都是基于“https”协议的，地址如下：

https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b

然后打开“PostMan”工具，如果没有安装“PostMan”，也可以在网上找一个在线的模拟请求工具网页进行测试。在“PostMan”中，选好“Post”的请求形式并粘贴好节点链接。这里，你可能会有一个疑问，为什么直接使用“Post”这个“RESTful API”的请求方式来进行测试，不是应该使用“RPC”吗？

原因是以太坊的源码中提供了可以选择是否开启“Http”服务选项的功能，而 Infura 的节点都是开启了的，所以我们可以方便地使用“RESTful API”的形式进行访问，如图 4-17 所示。

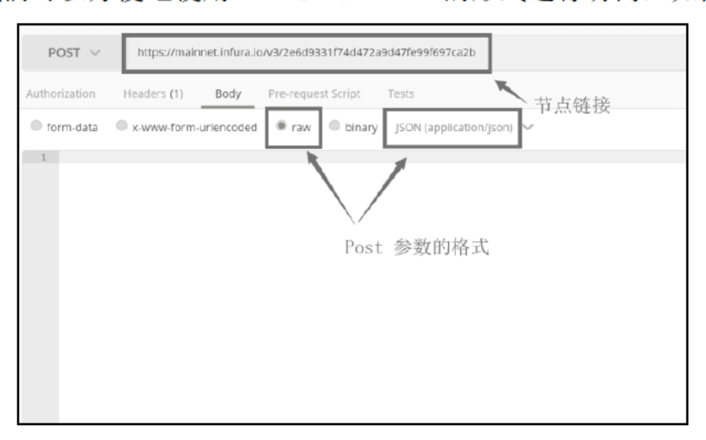



图 4-17　使用 PostMan 工具测试 Infura 提供的节点链接

再到区块链浏览器“https://etherscan.io/”中随便找一笔交易，复制它的哈希值，准备到“PostMan”中使用以太坊的“eth_getTransactionByHash”查询这笔交易的详情，如图 4-18 所示。

```
0x7be00cb83d7a3bda70fb8bd06142e6bb121bcdf2f665839d9a65671f5cacb098
```

Transaction Details　Earn Interest

Etherscan - Sponsored slots available. Book your slot here!

Overview　Event Logs (1)　State Changes New　Comments

| | |
|---|---|
| Transaction Hash: | 0x7be00cb83d7a3bda70fb8bd06142e6bb121bcdf2f665839d9a65671f5cacb098 |
| Status: | Success |
| Block: | 6614660　1156171 Block Confirmations |
| Timestamp: | 197 days 9 hrs ago (Oct-31-2018 01:12:45 AM +UTC) |
| From: | 0x5b3d576c6e828bd51470150847b106032eba9e25 |
| To: | Contract 0x78021abd9b06f0456cb9db95a846c302c34f8b8d |
| Tokens Transferred: | From 0x5b3d576c6e8... To 0x37f06ad5050... For 10,000 ($1.46) ERC-20 (CDCC) |

图 4-18　查询交易详情

回到“PostMan”中，进行请求参数的组装。按照“Json-RPC”的参数格式，组装好如下的参数（参考图 4-19）：

```
{
    "jsonrpc": "2.0",
    "method": "eth_getTransactionByHash",
    "params": [
"0x7be00cb83d7a3bda70fb8bd06142e6bb121bcdf2f665839d9a65671f5cacb098"
    ],
    "id": 23
}
```




图 4-19　组装好请求参数

然后单击“Send”按钮，即可进行访问，如图 4-20 所示。

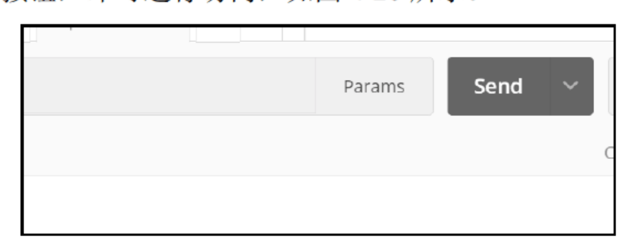



图 4-20　单击 PostMan 工具中的发送网络请求按钮

可以看到结果返回了我们要查询的交易的详细信息。由于返回的数据是十六进制格式的字符串，稍做转化后再和上面区块链浏览器 https://etherscan.io/中的数据进行对比，就会发现数据是一致的，这也就证明了 Infura 的链接是可以使用的，如图 4-21 所示。

图 4-21　PostMan 工具在请求后显示返回的交易数据

## 4.4.5　获取测试币

上面一节我们进行的是主网上的非转账类测试，如果要进行转账，就会产生真实的公链上的资产转移。因此一般资产方面的开发测试都是在测试网络中进行的，测试网络有自行搭建的测试节点网络和现在大型公用的 ROPSTEN、KOVAN 和 PINKEBY 测试网络。下面我们以 ROPSTEN 为例来测试获取以太坊 ETH 代币。

首先打开“水龙头”网站 https://faucet.ropsten.be/。Ropsten 测试以太坊 ETH 代币主要是从这个网站进行申请。进入主页后，直接在输入框中输入要接收测试币的以太坊钱包地址，再单击“Send me test Ether”按钮即可，如图 4-22 所示。




图 4-22　在水龙头网站获取测试币

在按钮下面显示出了交易的哈希值，证明已经给我们的地址发送了测试的以太币，如图 4-23 所示。目前在水龙头网站中每个地址每天只能领取一次测试以太币，如果想在短时间内获取多个，可以使用多个钱包地址来领取，然后在测试网络中通过以太坊转账交易来汇总。

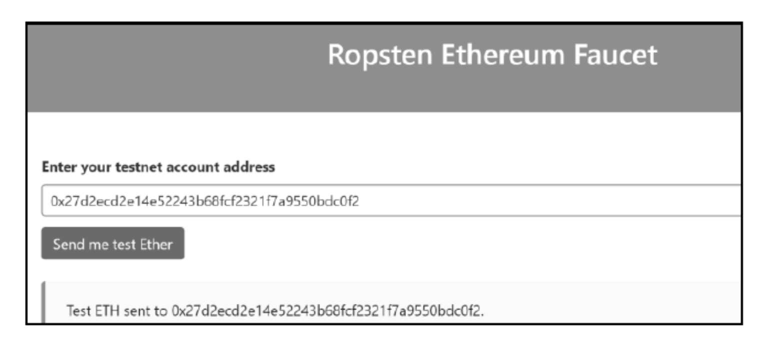



图 4-23　显示交易的哈希值

领取了测试以太币，我们到 Ropsten 测试网络的区块链浏览器网页中输入地址“0x27D2ECD2E14e52243b68FcF2321f7a9550bdc0f2”，对以太坊 ETH 余额查询进行验证。此外，其他测试网络产生的交易信息也都可以到网页中进行查询，链接为 https://ropsten.etherscan.io/address/0x27d2ecd2e14e52243b68fcf2321f7a9550bdc0f2，如图 4-24 所示。

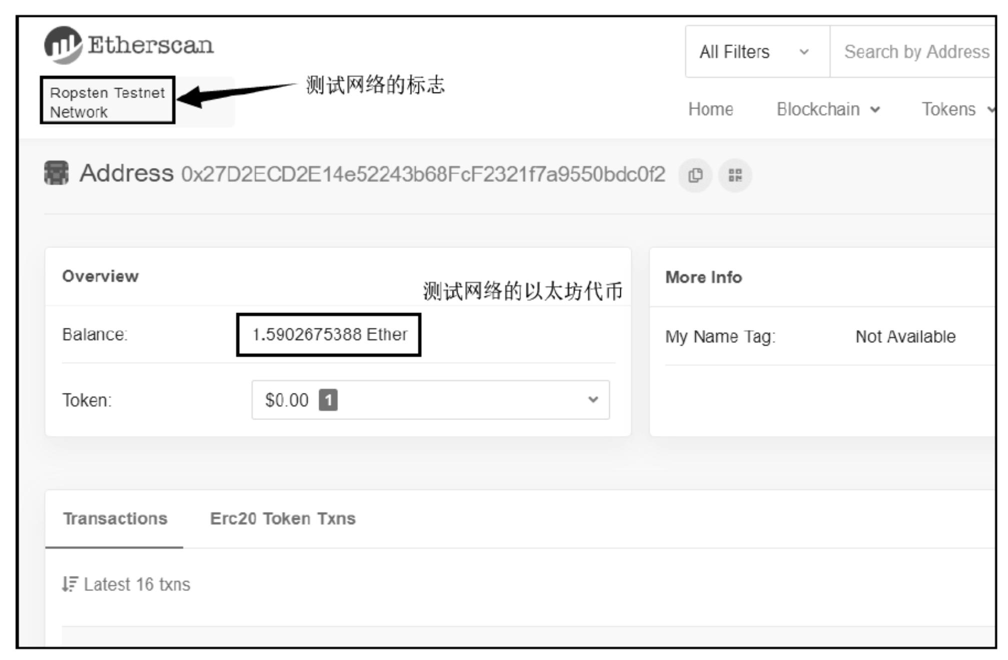



图 4-24　在区块链浏览器查看获取了的测试币

# 4.5　项目准备

在对中继有了整体的认识并获取了节点链接的相关准备信息后，我们开始介绍“以太坊中继”代码编写的项目准备。

本项目将会基于下面的环境及编辑器：

（1）使用“Golang”开发语言。

（2）编译环境是“go1.9.2”，也可以选择更高版本。

（3）开发工具是“Goland”。

“Go”编译环境在电脑上的配置，这里不做过多说明。对于不同操作系统的电脑，读者可以到网上搜索相关的教程。例如，“Windows 64”系统可以参考下面的链接：

https://blog.csdn.net/kwame211/article/details/79094695

开发工具“Goland”的下载地址是 https://www.jetbrains.com/go/?fromMenu。打开链接后，单击“DOWNLOAD”按钮进行下载，如图 4-25 所示。

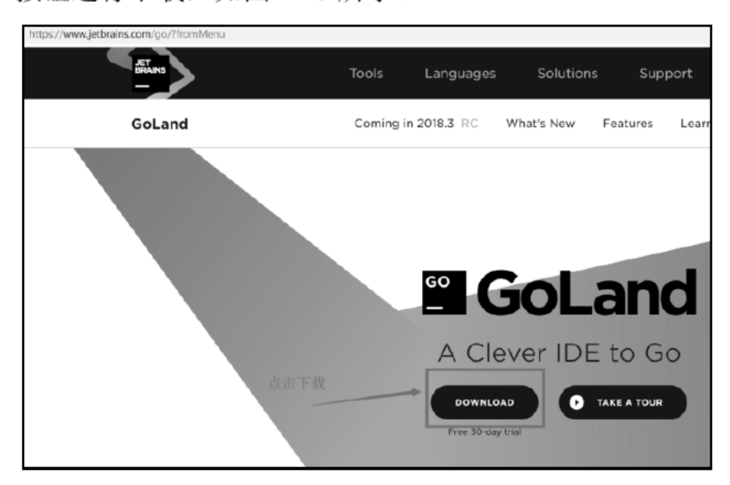



图 4-25　Goland 开发工具的下载页面

下载好之后，按照常规的软件安装流程进行安装。在欢迎页面直接单击“Next”按钮。在软件的安装路径页面中，也可以直接采取默认设置，单击“Next”按钮，按以下流程操作即可，如图 4-26 所示。



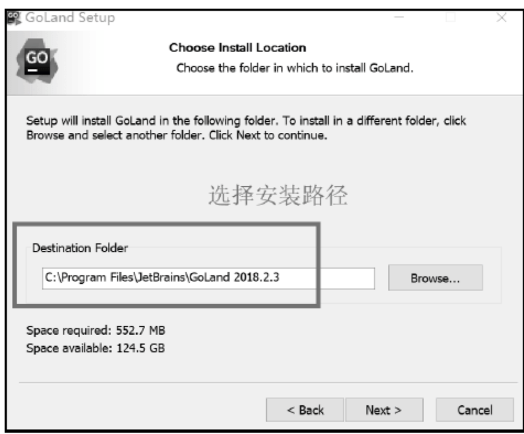




图 4-26　安装 Goland 的流程

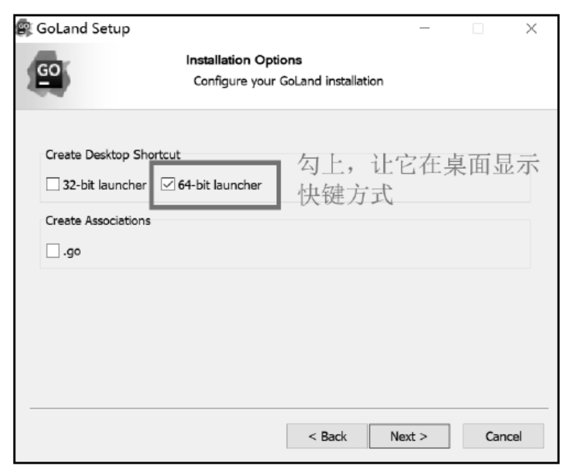












图 4-26（续）

选择了“Do not import settings”后，直接单击“OK”按钮。在看到的界面用鼠标滑动到最下面，再单击“Accept”按钮。在最后的激活页面，目前网上也有一些可用的激活码，如果没找到可用的，可先选择“30 天免费”使用，最后单击“Evaluate”按钮进行激活，如图 4-27 所示。




图 4-27　选择 30 天免费的界面

# 4.6　创建项目

接下来开始创建以太坊中继项目。首先进入到我们配置“Go”编译环境的环境变量“GOPATH”所在的文件夹，例如电脑的路径是“D:\go_1.9\go_path”，就进入到“go_path”文件夹，如图 4-28 所示。

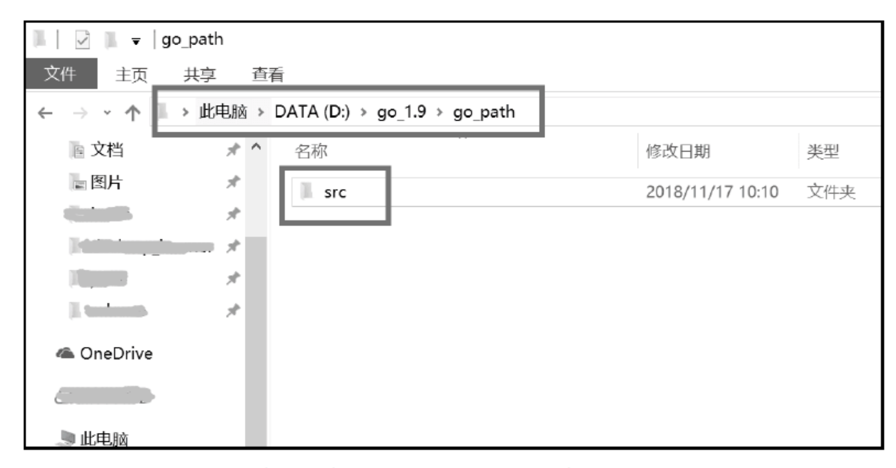



图 4-28　创建 Go 编译环境的 GOPATH 变量所指向的文件夹

在文件中创建一个名称为“src”的文件夹，用来作为后续项目的源代码目录（文件夹）。

“src”文件夹将会作为一个我们所有“go”项目的父级文件夹。现在我们进入到“src”文件夹中，创建“以太坊中继”项目“eth-relay”，如图 4-29 所示。




图 4-29　在 go_path 下创建项目

现在我们回到安装好的“Goland”开发工具中，在执行完了“4.4 项目准备”一节的最后一个步骤后，“Goland”会自动打开主页面。

在主页面中，先单击左上角的“File”菜单项，再单击“Open”菜单选项，意图是打开一个项目，在单击“Open”之后所弹出的选择框中找到我们刚刚创建的“eth-relay”项目，然后打开它，如图 4-30 所示。

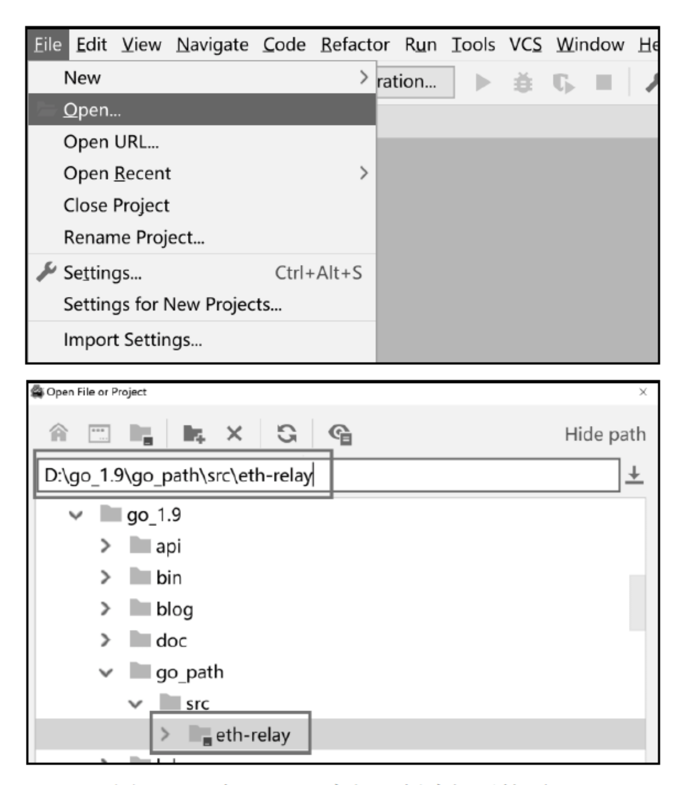



图 4-30　在 Goland 中打开创建好了的项目

打开后如果看到的是如图 4-31 所示的页面，就说明打开成功了。

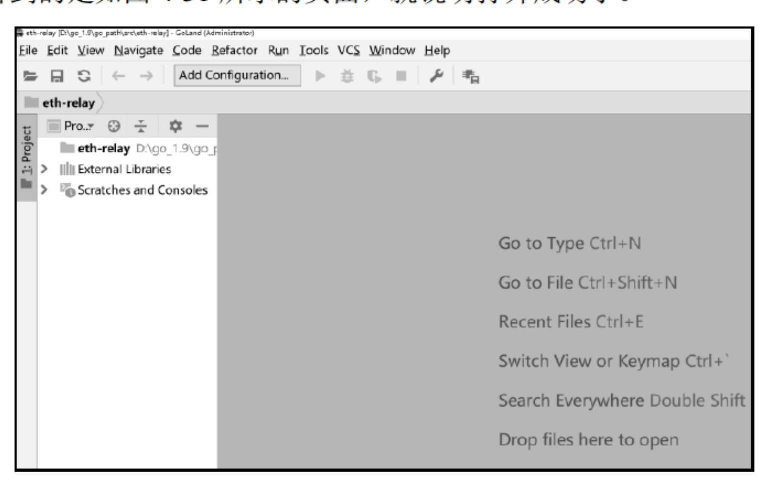



图 4-31　Goland 中打开创建好了的项目

接下来，在“Goland”中设置“go”的环境变量。首先还是单击左上角的“File”菜单项，然后单击“Settings”菜单选项，进入到设置页面后再单击列表栏中的第一项“Go”，进入到“Go”相关设置的页面，如图 4-32 所示。

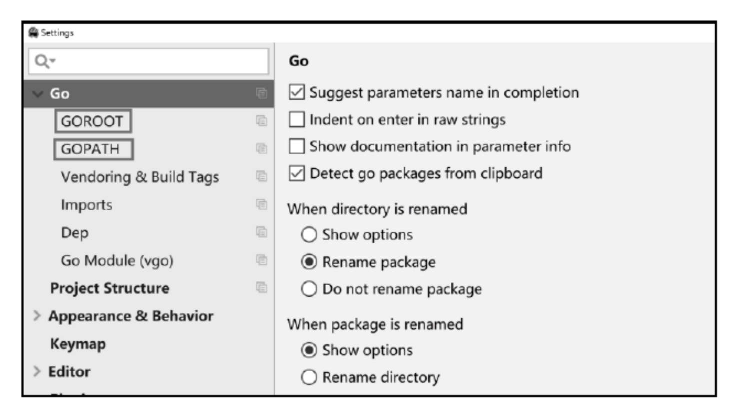



图 4-32　在 Goland 中设置 GOROOT 或 GOPATH 环境变量

单击图 4-32 中的“GOROOT”选项，进行“GOROOT”的设置；同理，对“GOPATH”选项进行相关的设置，本例保持不变即可。

为什么在一开始的时候下载安装并设置好了“Go”的编译环境后，在这里又要进行一次设置呢？这是因为在“Goland”开发工具中设置的环境变量，其优先级别是最高的，如果不进行设置，就会使用安装时的默认设置，这并不适合我们的要求。

单击“GOROOT”选项，可以看到此时“Goland”已经默认帮我们选择好了当初安装时设置的路径，如图 4-33 所示。

如果发现“GOROOT”还没有对应任何“Go”版本的路径，或者想进行自定义修改，需要我们手动选择添加，如图 4-34 所示。

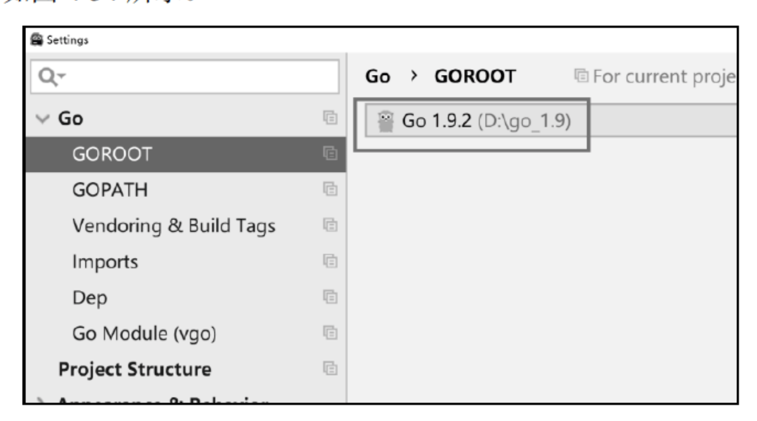



图 4-33　Goland 默认选择的 GOROOT 环境变量文件夹

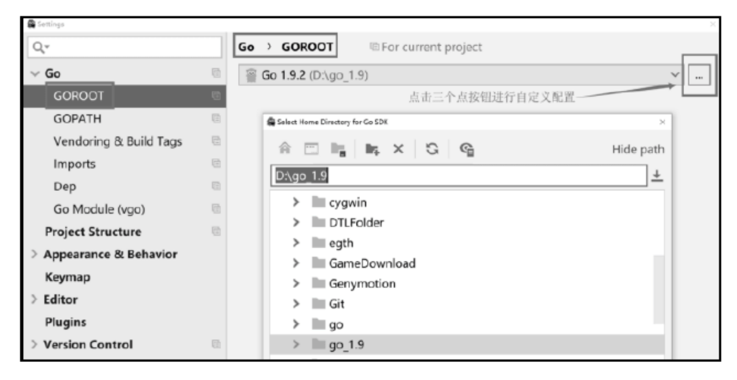



图 4-34　手动设置 GOROOT 环境变量文件夹

同样地，“GOPATH”选项也可以通过在“Goland”中查看当前“GOPATH”环境变量的文件夹来进行设置，即可以在图 4-35 中进行有关设置。

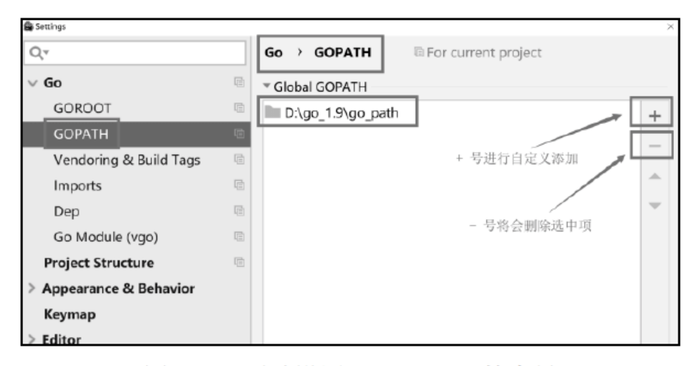



图 4-35　手动设置 GOPATH 环境变量

# 4.7　第一个 Go 程序

在“GOTOOT”和“GOPATH”环境变量都设置好之后，就可以进行项目的代码编写了。退出所有的设置界面，回到我们刚刚打开了的“eth-relay”项目的主页面中。

首先，将鼠标移动到界面中的项目名称处，用鼠标右击选择“New”→“GoFile”，创建一个新的“Go”文件，取名为“main.go”，如图 4-36 所示。

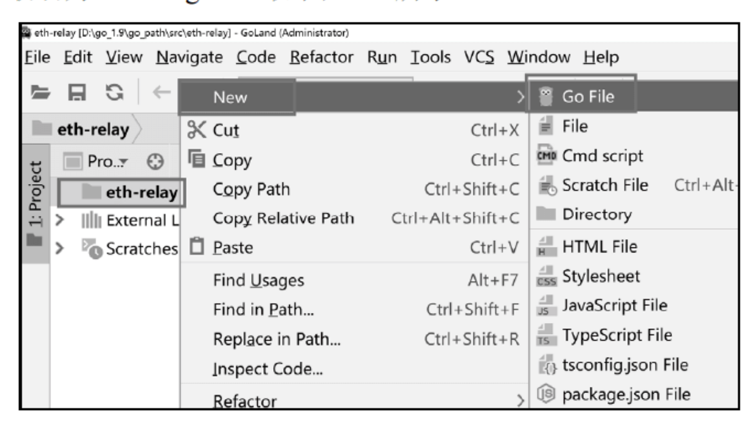



图 4-36　在项目文件夹下创建一个 Go 的代码文件

可以看到“eth-relay”项目文件夹下多出了一个“main.go”文件。“.go”后缀结尾的就是“go”语言的标准源代码文件，用鼠标双击文件就能在编辑器的右边区域进行代码编辑，如图 4-37 所示。

接下来我们先编写好下面的一段测试代码，尝试编译一下，如图 4-38 所示。代码运行后将会在“Goland”自带的控制台中输出“hello ETH”字符串。先在“main.go”文件中将第一行的“package eth_relay”改为“package main”，表示这个“main.go”文件将是我们整个程序启动时的入口文件，而后再输入代码。其中，代码中出现的“/** 内容 */”代表的是注释部分，可以将其删除。

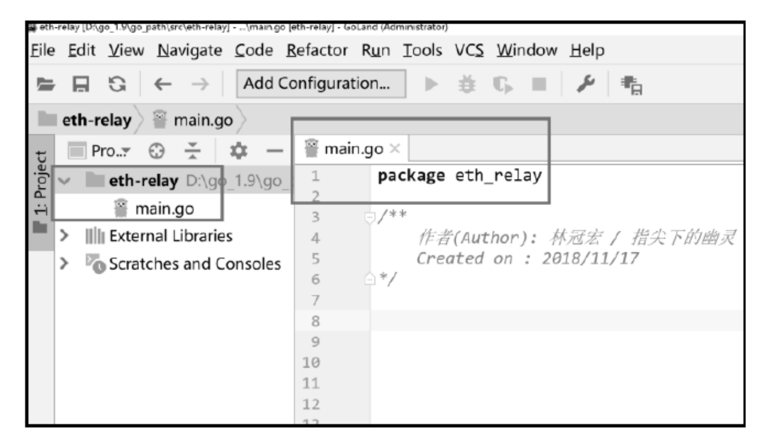



图 4-37　创建好了的代码文件显示的初始内容

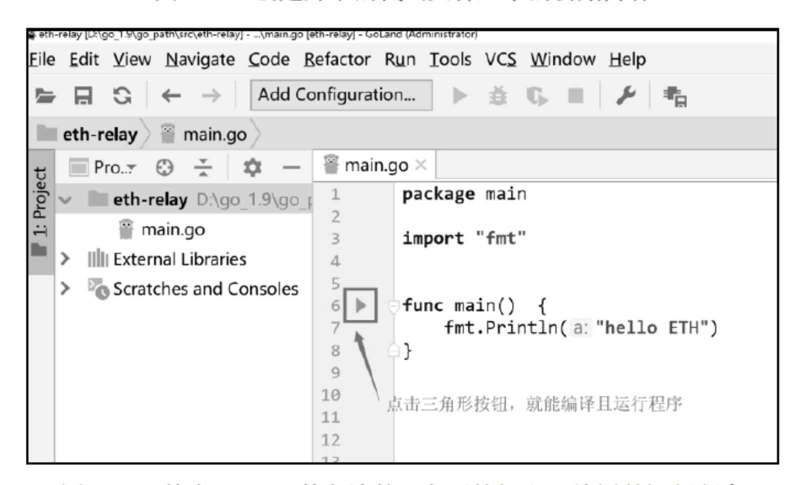



图 4-38　单击 main 函数左边的三角形按钮即可编译并运行程序

```
package main
import "fmt"
func main()  {
   fmt.Println("hello ETH")
}
```

在“Goland”中，默认会自动进行依赖包的导入，例如上述代码中的“import "fmt"”就不需要我们手动输入。单击绿色的三角形按钮，在弹出的框中单击第一栏“run go build main.go”就能对整个程序进行编译并运行。运行的结果将会在控制台中输出，如图 4-39 所示。

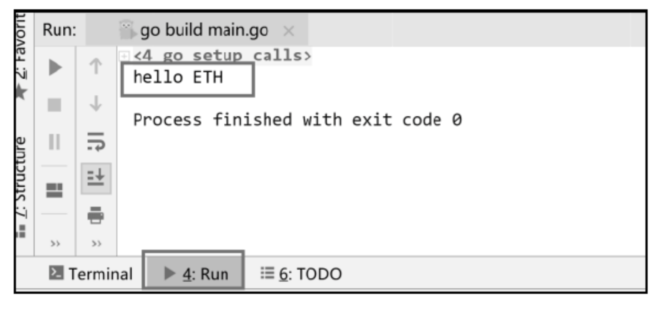



图 4-39　编译并运行程序后，在控制台输出的内容

# 4.8 封装“RPC”客户端

在中继服务端中所充当的请求客户端角色，其作用是在我们每次请求以太坊节点接口时提供请求的句柄，它的入参应该对应节点链接，如图 4-40 所示。

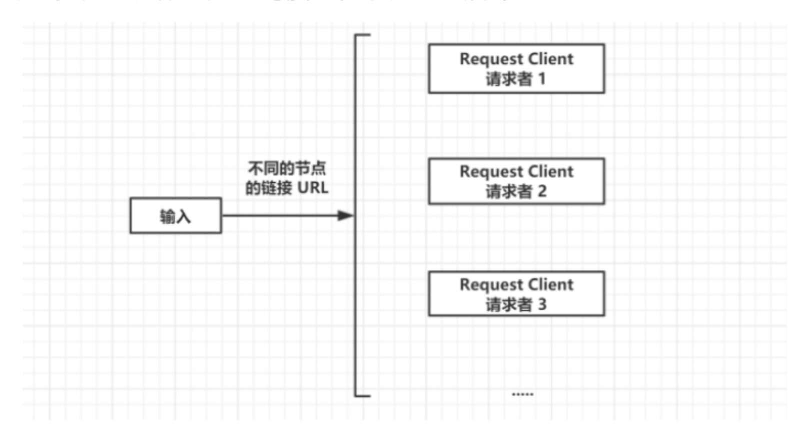



图 4-40　不同节点链接对应的请求客户端

前面谈到，我们访问以太坊节点的接口都是“RPC”接口类型，这就要求我们的客户端属于“RPC”客户端，首先新建一个名为“ETH_RPC_Client.go”文件，如图 4-41 所示。

在接下来的过程中，“RPC”客户端内部将会用到以太坊源码中的一个“RPC”依赖库。为什么使用以太坊的依赖库？这个并不是强制要求的，也可以使用其他的依赖库。建议读者在实际开发中使用到各种功能库时，如果能从以太坊 go 版本源码获取，就尽量不要去引用其他的库，或者自己进行库的编写。

下面我们来介绍依赖库的下载和使用。

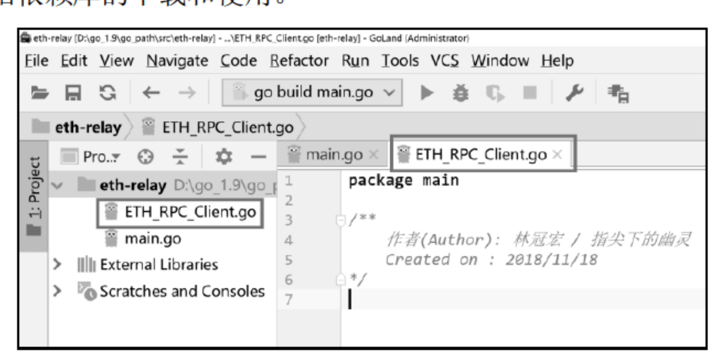



图 4-41　创建以太坊 RPC 客户端代码文件

## 4.8.1 下载依赖库

以太坊 go 版本源码自带的依赖库在“GitHub”上，是开源的，读者需要的话，就可以自己下载。可以使用“Go”编译环境自带的命令下载到“go path”文件夹中，下载好之后，就可以直接进行使用了，命令是：

```
go get xxxx
```

其中，xxxx 代表远程依赖库在“GitHub”中的路径，以太坊 go 版本源码的路径是 github.com/ethereum/go-ethereum。在“Goland”底部的控制台“Terminal”中输入下面的命令，按回车键，执行命令就可以进行下载了，如图 4-42 所示。

```
go get github.com/ethereum/go-ethereum
```




图 4-42　在 Goland 命令行控制台输入命令

下载过程中，需要等待一段时间，时间的长度取决于网速等因素，依赖库下载好之后，在控制台能够看到的现象是，命令行的“开始输入行”另起了一行，且没有错误信息输出。如图 4-43 所示。

```
Terminal
Microsoft Windows [版本 10.0.17134.407]
(c) 2018 Microsoft Corporation。保留所有权利。

D:\go_1.9\go_path\src\eth-relay>go get github.com/ethereum/go-ethereum

D:\go_1.9\go_path\src\eth-relay>      另起了一行
```

图 4-43　按回车键运行命令结束后控制台的显示

也可以到“go path”文件夹下验证查看，如果“go-ethereum”依赖库已经成功被下载下来，那么它将会存放于“github.com”文件夹中的“ethereum”中，如图 4-44 所示。

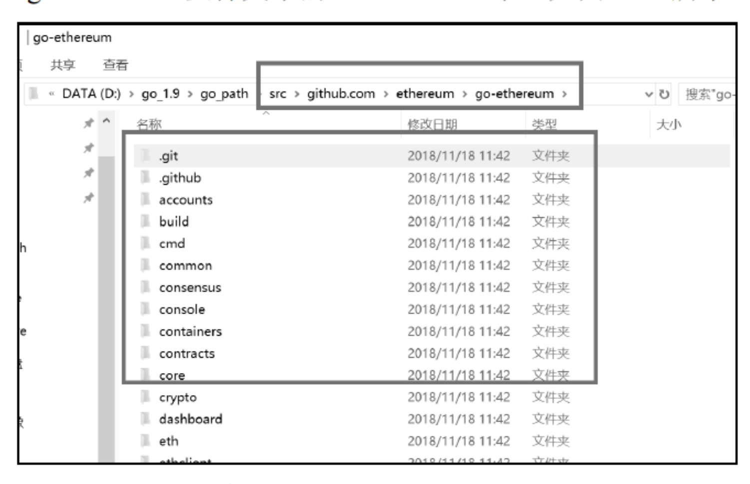



图 4-44　通过 Go 的 Get 命令下载好了的且存放在 GOPATH 文件夹下的以太坊源码

依赖库下载成功后，下面我们来定义并实现“RPC”客户端。

## 4.8.2 编写“RPC”客户端

在新建的“ETH_RPC_Client.go”文件中输入下面的代码。

```
type ETHRPCClient struct {
   NodeUrl string       // 代表节点的 url 链接
   client  *rpc.Client  // 代表 rpc 客户端句柄实例
}

// NewETHRPCClient 代表的是新建一个“RPC” 客户端
// 入参 nodeUrl 就是节点的链接，返回的是带有 * 的 ETHRPCClient 对象指针
func NewETHRPCClient(nodeUrl string) *ETHRPCClient {
   // & 符号代表的是取指针
   return &ETHRPCClient{
      NodeUrl:nodeUrl,
   }
}
```

在输入了上面的代码后，会发现在“client *rpc.Client”行中“rpc”显示的是红色字体，代表这里有错误。此时，用鼠标单击一下编辑框内的一个区域，就能看到“Goland”给予我们的错误提示，如图 4-45 所示。这个提示的意思是：依赖库 rpc 的名称有重复，需要手动选择导入，无法自动识别进行包的导入。

提示语中有一个“Alt+Enter”组合键文字，在提示框还没有消失的时候，按下这个组合键，就能进入到选择库的弹框中，在选择库的弹框里面可以手动进行库的选择，如图 4-46 所示。共有两个“RPC”库。第一个是“net/rpc”，代表“Go”环境安装时自带的系统网络库；第二个是“go-ethereum”，也就是我们前面所下载的以太坊 go 版本节点源码中的依赖库。理所当然，我们要选择第二个。

图 4-45　Goland 提示可以通过组合键 Alt +Enter 把依赖包导入到代码中

```
type ETHRPCClient struct {
    NodeUrl string
    client  *rpc.Client
}

func NewETHRPCClient(nc
    return &ETHRPCClient{
        NodeUrl:nodeUrl,
    }
}
```

Package to import
net/rpc
github.com/ethereum/go-ethereum/rpc

在第一个弹框显示时，按下ALT+Enter组合键，进入到选择库的弹框中，此时按键盘的上下键或用鼠标单击就能选择相应的库

图 4-46　多个同名的依赖包，需要选择导入哪一个

手动选择好了依赖库后，可以看到代码中多出了导入包的一行，如图 4-47 所示。

main.go × ETH_RPC_Client.go ×

```
1   package main
2
3   import (
4       "fmt"
5       "github.com/ethereum/go-ethereum/rpc"
6   )
7
8   type ETHRPCClient struct {
9       NodeUrl string       // 代表节点的url链接
10      client  *rpc.Client  // 代表 rpc 客户端句柄实体
11  }
12
13  // NewETHRPCClient 代表的是新建一个“RPC”客户端
14  // 入参 nodeUrl，就是节点的链接，返回的是带有 * 号的ETHRPCClient指针对象
15  func NewETHRPCClient(nodeUrl string) *ETHRPCClient {
16      // & 符号代表的是取指针
17      return &ETHRPCClient{
18          NodeUrl:nodeUrl,
19      }
20  }
```

图 4-47　导入依赖包后代码文件显示出对应的路径

接下来我们实现初始化“*rpc.Client”句柄实例，请将必要的函数补充到“ETH_RPC_Client.go”中。到了这一步，我们的“RPC”客户端的封装差不多就完成了。代码如下：

```
// 初始化 rpc 客户端句柄实例
func (erc *ETHRPCClient) initRpc() {
   // 使用 go-ethereum 库中的 rpc 库来初始化
   // DialHTTP 的意思是使用 http 版本的 rpc 实现方式
   rpcClient,err := rpc.DialHTTP(erc.NodeUrl)
   if err != nil {
      // 初始化失败，终结程序，并将错误信息显示到控制台中
      errInfo := fmt.Errorf("初始化 rpc client 失败%s",err.Error()).Error()
      panic(errInfo)
   }
   // 初始化成功，将新实例化的 rpc 句柄赋值给 ETHRPCClient 结构体里面的 client
   erc.client = rpcClient
}
```

对于“initRpc()”初始化函数，将会在“RPC”客户端创建函数时进行调用，也就是在“NewETHRPCClient”函数内部进行调用，这里我们先修改“NewETHRPCClient”函数为下面的样子。

```
func NewETHRPCClient(nodeUrl string) *ETHRPCClient {
   // & 符号代表的是取指针
   client := &ETHRPCClient{
      NodeUrl:nodeUrl,
   }
   client.initRpc()  // 进行初始化 rpc 客户端句柄实例
   return client
}
```

最后，还要补充一个“GetRpc”函数，以便让“ETHRPCClient”结构体里面的 rpc 请求句柄实例“client”能够被外部引用。

```
// Go 语言语法中，大写字母开头的变量或者函数（方法）才能够被外部引用
// 小写字母的变量或函数（方法）只能内部调用
// GetRpc 函数（方法）是为了方便外部能够获取 client  *rpc.Client，以便进行访问
func (erc *ETHRPCClient) GetRpc() *rpc.Client {
   if erc.client == nil {
      erc.initRpc()
   }
   return erc.client
}
```

完整的代码如图 4-48 所示。

```
main.go × | ETH_RPC_Client.go ×
1   package main
2
3   import (
4       "fmt"
5       "github.com/ethereum/go-ethereum/rpc"
6   )
7
8   type ETHRPCClient struct {
9       NodeUrl string       // 代表节点的url链接
10      client  *rpc.Client  // 代表 rpc 客户端句柄实体
11  }
12
13  // NewETHRPCClient 代表的是新建一个“RPC” 客户端
14  // 入参 nodeUrl，就是节点的链接，返回的是带有 * 号的ETHRPCClient指针对象
15  func NewETHRPCClient(nodeUrl string) *ETHRPCClient {
16      // & 符号代表的是取指针
17      client := &ETHRPCClient{
18          NodeUrl:nodeUrl,
19      }
20      client.initRpc()  // 进行初始化 rpc 客户端句柄实体
21      return client
22  }
23
24  // 初始化 rpc 请求句柄实体
25  func (erc *ETHRPCClient) initRpc() {
26      // 使用 go-ethereum 库中的 rpc 库来初始化
27      // DialHTTP 的意思是使用 http 版本的 rpc 实现方式
28      rpcClient,err := rpc.DialHTTP(erc.NodeUrl)
29      if err != nil {
30          // 初始化失败，终结程序，并将错误信息显示到控制台上面
31          errInfo := fmt.Errorf( format: "初始化 rpc client 失败%s",err.Error()).Error()
32          panic(errInfo)
33      }
34      // 初始化成功，将新实例化的 rpc 句柄赋值给我们 ETHRPCClient 结构体里面的
35      erc.client = rpcClient
36  }
37
38  // Go 语言语法中，大写字母开头的变量或者方法才能够被外部引用
39  // 小写字母的变量或方法只能内部调用
40  // GetRpc 方法是为了方便外部能够获取 client  *rpc.Client 来进行访问
41  func (erc *ETHRPCClient) GetRpc() *rpc.Client {
42      if erc.client == nil {
43          erc.initRpc()
44      }
45      return erc.client
46  }
47
```

图 4-48　完整的代码

## 4.8.3　单元测试

单元测试在软件编写的过程中是必不可少的，因此在我们所要实现的“以太坊中继”中也需要对某些功能代码进行单元测试。

我们先对“RPC”客户端的初始化函数进行一轮简单的单元测试，这里使用的是“Go”语言自带的单元测试模块。

首先在项目中新建一个名称为“ethrpc_test.go”的文件，“Go”的单元测试模块规定了属于单元测试“go”编译文件的名称须符合后缀是“_test.go”，例如“nihao_test.go”，如图 4-49 所示。

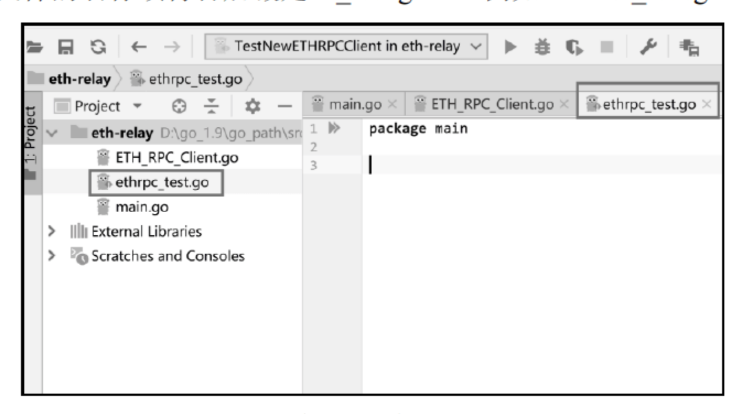



图 4-49　创建 Go 的单元测试代码文件

单元测试文件入口函数的定义规则是，以大写的“Test”单词开头。当然，“Goland”对于在单元测试文件内打出的字母“test”也会给予提示，如图 4-50 所示。在提示出现时直接按回车键就能自动补充为完整的单元测试函数。




图 4-50　Goland 的代码提示与自动补全

图 4-50 中有两点比较重要：第一点是每个自动补全的单元测试函数都是默认的“TestName”，这里需要我们自行进行名称的修改；第二点是，每个单元测试函数的左边都会显示一个绿色的△按钮，单击这个按钮即可运行当前的函数。

紧接着我们把“TestName” 修改为“TestNewETHRPCClient”，这样修改的意思是为了测试“NewETHRPCClient”这个函数，再输入如下的代码：

```
func TestNewETHRPCClient(t *testing.T) {
    // 首先是一个格式正确的链接测试初始化
    client2 := NewETHRPCClient("www.nihao.com").GetRpc()
    if client2 == nil {
        fmt.Println("初始化失败")
    }
    // 接着是 123://456 非法链接测试初始化
    client := NewETHRPCClient("123://456").GetRpc()
    if client == nil {
        fmt.Println("初始化失败")
    }
}
```

上面的代码一共测试了两个链接。最后单击“Goland”顶部工具栏的绿色△按钮进行编译及运行，观察控制台中的运行结果。

运行结果如图 4-51 所示。可以看到，在解析链接“123://456”的时候出现了错误，表明这不是一个合法的链接。由于是初始化阶段的错误，因此我们直接采用“panic”的方式让程序崩溃，此时的错误相当于致命错误。如果不想让程序崩溃，可以在“ETH_RPC_Client.go”文件中将“panic(errInfo)”一行代码修改为别的内容。




图 4-51　控制台在代码运行错误时的红色提示信息

# 4.9　编写访问接口代码

接下来开始使用我们前面封装好的“RPC”客户端来对以太坊节点的“RPC”接口进行访问。接口在名称上的分类主要有两种：一种是参数不需要用到“data”字段的，另一种是需要用到

“data”字段的。例如，获取 ERC20 代币的余额，就需要设置好调用“eth_call”函数（方法）的“data”参数字段，而“eth_getTransaction”获取交易信息则不需要传递“data”参数。

## 4.9.1　认识“Call”函数

标准的“RPC”客户端都应该具备发出请求函数的功能，由于我们所使用的“RPC”依赖库基于以太坊“Go”版本源码，因此自然地依赖库就具备了请求函数。

在“Goland”中，对于在“Go”源码文件中所引入的依赖库文件，可直接通过按住 Ctrl 键再单击鼠标左键，即可进入到对应源码文件中查看其中的函数。

现在我们进入“ETH_RPC_Client.go”源码文件所引用的“RPC”库文件中，按住 Ctrl 键再用鼠标左键单击源码文件下面一行代码最后边的“rpc”单词。

```
"github.com/ethereum/go-ethereum/rpc"
```

可以看到，在编辑器的主页面中显示出了“rpc”依赖包下的源码文件文件夹，其中有一个“client.go”文件，如图 4-52 所示。

图 4-52　在 Goland 的代码文件中单击导入后的依赖包名称可以看到对应的源码文件夹

用鼠标双击“client.go”文件，在打开的文件中可以看到每个函数的注释，如图 4-53 所示。

图 4-53　在打开的源码文件夹中双击源码文件来查看源码的内容

通过查看“client.go”源码文件，发现其中有一个名称为“Call”的函数（如图 4-54 所示），该函数允许调用者发起“RPC”请求，这正是我们要找的函数。这个函数非常重要，可以说我们所要实现的绝大部分的以太坊“RPC”接口请求函数都会依赖“Call”函数来达到目的。下面逐个解析该函数所传入的参数含义。

```
238 // Call performs a JSON-RPC call with the given arguments and unmarshals into
239 // result if no error occurred.
240 //
241 // The result must be a pointer so that package json can unmarshal into it. You
242 // can also pass nil, in which case the result is ignored.
243 func (c *Client) Call(result interface{}, method string, args ...interface{}) error {
244     ctx := context.Background()
245     return c.CallContext(ctx, result, method, args...)
246 }
247
```

图 4-54　client.go 源码文件中的 Call 函数代码

“Call”函数如下：

```
func (c *Client) Call(result interface{}, method string, args ...interface{}) error
{
   ctx := context.Background()
   return c.CallContext(ctx, result, method, args...)
}
```

第一个参数“result”的数据类型是“Go”语言中的“interface{}”类型，它类似 Java 语言中的泛型，可以代表任何类型，作用是接收存储“RPC”请求后得到的结果值。

第二个参数“method”的数据类型是“string”字符串类型，作用是用来标明请求的接口名称，例如获取交易记录的“eth_getTransactionByHash”就是“method”一个可能的取值。

第三个参数“args”的数据类型是“Go”的多泛型参数“...interface{}”，表示可以传入一个参数或者两个、三个参数等，且每个参数都是一个“interface{}”泛型，既可以是 int64 整数类型，也可以是 string 字符串类型。

## 4.9.2　查找请求的参数

在项目中新建一个名称为“ETH_RPC_Requester.go”的文件，代表使用“RPC”客户端的请求者，此后所要进行封装的、访问以太坊接口的函数都将写在这个文件中。

以下编写根据以太坊交易哈希值来获取交易信息的接口函数，作为示例来介绍接口函数的实现方法。

首先编写下面的代码，进行“RPC”请求者实例化的定义及其函数的初始化。因为每个请求者都需要拥有一个客户端成员，一一对应，请求者从自身绑定的“RPC”客户端发出请求，所以请求者“ETHRPCRequester”内部拥有一个客户端指针类型的变量“client *ETHRPCClient”。

```
type ETHRPCRequester struct {
   client *ETHRPCClient  // 小写字母开头，私有的 rpc 客户端
}

func NewETHRPCRequester(nodeUrl string) *ETHRPCRequester {
   requester := &ETHRPCRequester{}
```

```
    // 实例化 rpc 客户端
    requester.client = NewETHRPCClient(nodeUrl)
    return requester
}
```

代码如图 4-55 所示。

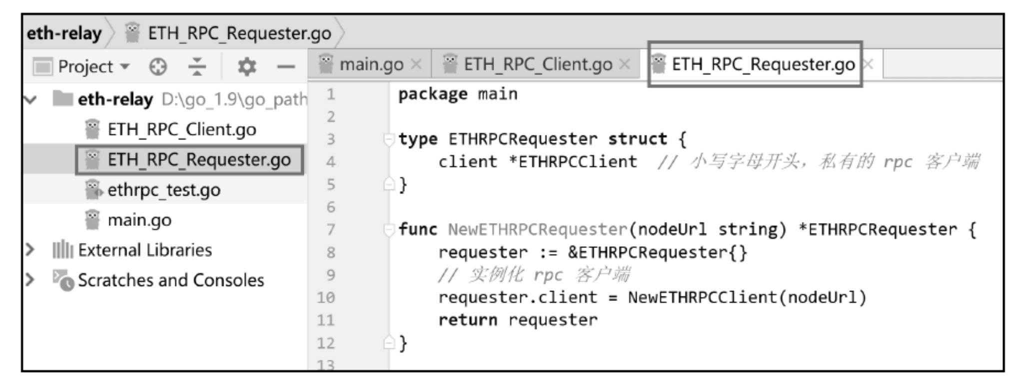



图 4-55　ETH_RPC_Requester.go 代码

接下来实现“获取交易信息”接口函数。根据前文对客户端“Call”函数入参的介绍，“获取交易信息”接口函数的 3 类参数，乃至之后要实现的访问其他以太坊“RPC”接口函数的参数，都可以根据下面的两种方式得知：

（1）查看官方文档，相关信息在“以太坊接口”一节中已经介绍过。

（2）查看本书曾列举并介绍过的接口。

在“获取交易信息”接口函数中，查看官方“RPC”接口文档，其链接是 https://ethereum.gitbooks.io/frontier-guide/。打开文档后，可以看到以太坊所有相关的介绍都完全具备。在主页中找到第 4 小节中的“JSON RPC API”，单击即可，如图 4-56 所示。

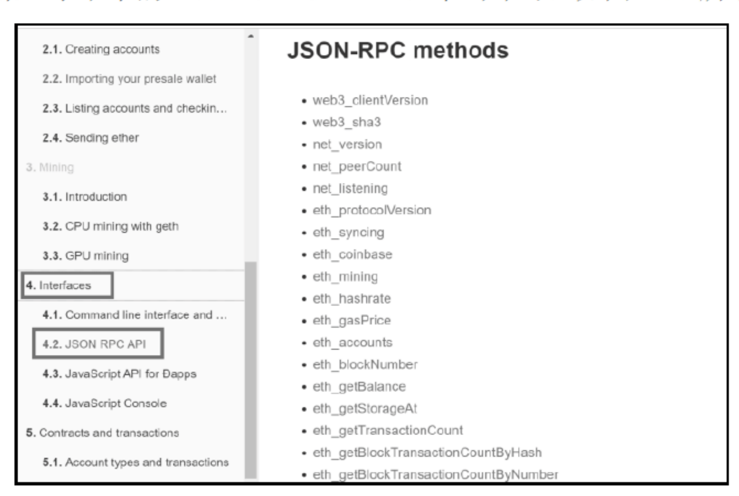



图 4-56　在接口文档中查看接口信息

在左边的“methods”列表中找到“eth_getTransactionByHash”的介绍，便可以很简单地获取我们需要的信息，如图 4-57 所示。




图 4-57 eth_getTransactionByHash 接口的文档介绍页面

“Parameters”对应的是参数列表，可以看到只需要传入一个 32 字节的交易哈希值，类型是一个字符串。“Returns”对应的是返回值列表及其介绍，而“method”就是第一行中的“eth_getTransactionByHash”。

至此，通过查看文档，我们把要实现的“获取交易信息”接口函数所需要的参数都找到了，接下来开始编写代码。

## 4.9.3 实现获取交易信息

本节我们在“ETH_RPC_Requester.go”文件中完成获取交易信息接口的编写。从上面的一节中可以知道，交易信息接口对应的“method”是“eth_getTransactionByHash”，接下来我们在项目中新建一个名称为“model”的文件夹，专门用来存放数据结构体。

在“Goland”中新建文件夹的方式如图 4-58 所示，将鼠标停放到“eth_relay”文件夹名称上，单击鼠标右键即可看到对应的选项列表，然后单击“New”→“Directory”选项。

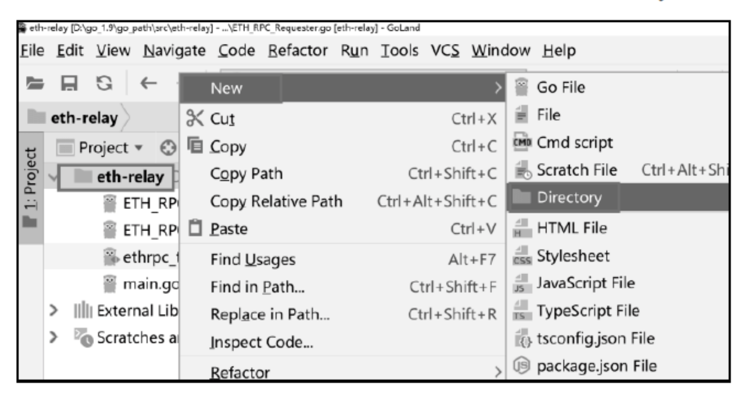



图 4-58 在项目文件夹中创建另一个文件夹

“model”文件夹创建好之后，在里面创建一个名称为“transaction.go”的文件夹，准备编写

交易信息数据所对应的变量值，如图 4-59 所示。

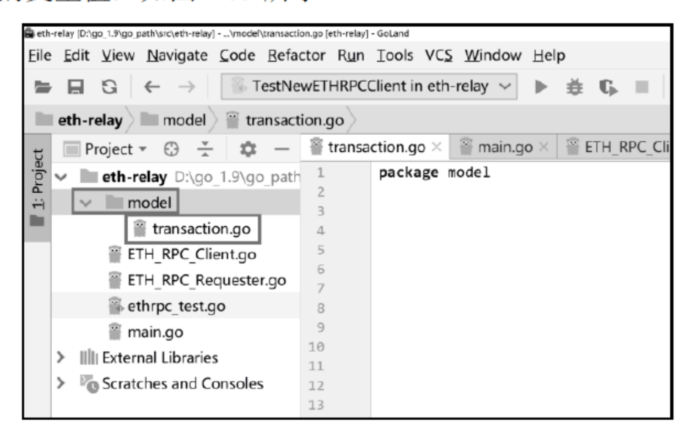



图 4-59　创建 transaction.go 文件

根据“eth_getTransactionByHash”接口对应的返回值，我们逐个在“transaction.go”文件中编写其对应的数据类型。在数据变量类型映射部分，文档中所有标明的无论是“Bytes”“DATA”还是“QUANTITY”，都可以使用“Go”语言中的“string”字符串类型与之对应。因为即使是数字返回值，其格式实际上也是一个十六进制的字符串，如图 4-60 所示。

**Returns**

`Object` - A transaction object, or `null` when no transaction was found:

- `hash`: `DATA`, 32 Bytes - hash of the transaction.
- `nonce`: `QUANTITY` - the number of transactions made by the sender prior to this one.
- `blockHash`: `DATA`, 32 Bytes - hash of the block where this transaction was in. `null` when its pending.
- `blockNumber`: `QUANTITY` - block number where this transaction was in. `null` when its pending.
- `transactionIndex`: `QUANTITY` - integer of the transactions index position in the block. `null` when its pending.
- `from`: `DATA`, 20 Bytes - address of the sender.
- `to`: `DATA`, 20 Bytes - address of the receiver. `null` when its a contract creation transaction.
- `value`: `QUANTITY` - value transferred in Wei.
- `gasPrice`: `QUANTITY` - gas price provided by the sender in Wei.
- `gas`: `QUANTITY` - gas provided by the sender.
- `input`: `DATA` - the data send along with the transaction.

图 4-60　eth_getTransactionByHash 接口的返回值及其介绍

根据对应关系，可以编写出下面交易信息所对应的数据结构体的代码。其中，“json:”xxx””的意思是这个结构体可以被“Json”序列化以及被反序列化，而“xxx”代表的是当结构体被序列化输出的时候在“Json”格式下所对应的名称。

```
type Transaction struct {
  Hash              string    `json:"hash"`
  Nonce             string    `json:"nonce"`
  BlockHash         string    `json:"blockHash"`
  BlockNumber       string    `json:"blockNumber"`
  TransactionIndex  string    `json:"transactionIndex"`
  From              string    `json:"from"`
```

```
  To              string    `json:"to"`
  Value           string    `json:"value"`
  GasPrice        string    `json:"gasPrice"`
  Gas             string    `json:"gas"`
  Input           string    `json:"input"`
}
```

此外，“Transaction”数据结构体中的每个变量必须是大写开头，原因是只有这些变量大写时，“Transaction”结构体在文件外被实例化的时候才能被外部引用，可以理解为公有变量，例如像下面的引用形式：

```
   Tx := model.Transaction{}
   Tx.Hash = "0x123456789"
```

到了这里，当前的获取交易信息接口对应于发起“RPC”请求的“Call”函数所需的参数已经齐全了。

现在我们来完成“getTransactionByHash”函数，代码如下：

```
// 根据交易的哈希值获取对应交易的信息
func (r *ETHRPCRequester) GetTransactionByHash(txHash string)
(model.Transaction,error) {
  methodName := "eth_getTransactionByHash"
  result := model.Transaction{}
  // 下面 call 函数的 result 参数传入的是  model.Transaction 结构体的引用，
  // 这样内部所设置的值在函数执行完之后才能有效果
  err := r.client.GetRpc().Call(&result,methodName,txHash)
  return result,err
}
```

接下来我们使用在“获取链接”一节中所获得的主网以太坊节点链接（https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b）来进行获取交易记录的单元测试。

首先到以太坊区块链浏览器“https://etherscan.io/”上随便找一笔交易的哈希值，进行一下查询，如图 4-61 所示。

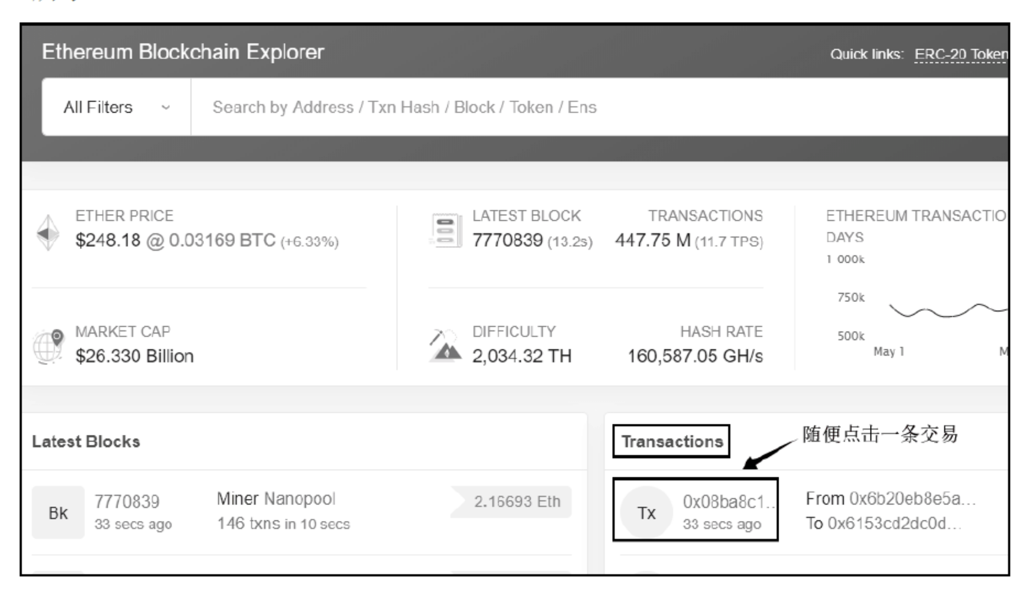


图 4-61　查询交易

要查询的是：

```
0x53c5b03e392d6aa68a0df26b6d466ae8fbd1c2c5b74f9baae05434ec9a18a282
```

它所对应的数据如图 4-62 所示。

图 4-62　交易数据

编写单元测试代码如下：

```
func Test_GetTransactionByHash(t *testing.T) {
  nodeUrl := "https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b"
  txHash  :=
"0x53c5b03e392d6aa68a0df26b6d466ae8fbd1c2c5b74f9baae05434ec9a18a282"
  if txHash == "" || len(txHash) != 66 {
    // 这里演示在调用 RPC 接口函数的时候，要先进行入参的合法性判断
    fmt.Println("非法的交易哈希值")
    return
  }
  txInfo,err := NewETHRPCRequester(nodeUrl).GetTransactionByHash(txHash)
  if err != nil {
    // 查询失败，打印出信息
    fmt.Println("查询交易失败，信息是：",err.Error())
    return
  }
  // 查询成功，将 transaction 结果的结构体以 json 格式序列化，再以 string 格式输出
  json,_ := json.Marshal(txInfo)
  fmt.Println(string(json))
}
```

运行后，观察控制台的输出结果，可以看到完整的交易数据已经按照“Json”格式输出了，再与浏览器中的显示对比，数据完全一致，这表示“获取交易信息”接口函数已经完成，如图 4-63 所示。

```
Run:  Test_GetTransactionByHash in eth-relay
Tests passed: 1 of 1 test – 1 s 63 ms
<4 go setup calls>
{"hash":"0x53c5b03e392d6aa68a0df26b6d466ae8fbd1c2c5b74f9baae05434ec9a18a282","nonce":"0x0"

Process finished with exit code 0
```


```
{
  "hash": "0x53c5b03e392d6aa68a0df26b6d466ae8fbd1c2c5b74f9baae05434ec9a18a282",
  "nonce": "0x0",
  "blockHash": "0x458ff523a4137ae7e7b60a96dec41e9af30072067f58f8dfdc64bc092bb10884",
  "blockNumber": "0x662a7d",
  "transactionIndex": "0x5b",
  "from": "0x9d84338d8503d95c351431e0fc705ed5d5793f5b",
  "to": "0x78021abd9b06f0456cb9db95a846c302c34f8b8d",
  "value": "0x0",
  "gasPrice": "0x1836e2100",
  "gas": "0xea60",
  "input": "0xa9059cbb00000000000000000000000037f06ad505036b55d961024789576790be180c0100000000000000000000000000000000000000000000000000000000021e19
}
```


图 4-63　Test_GetTransactionByHash 单元测试代码的运行结果

## 4.9.4　认识“BatchCall”函数

上面我们完成了一个单批次获取交易信息的接口函数，如果要实现根据多笔交易的哈希值来获取多个交易的信息，读者可能会想到，使用循环语法就能达到目的。例如，使用“for”来多次调用单批次的那个接口。这种使用循环的做法是可以的，但不是性能最好的。和“Call”函数（方法）一样，以太坊的“Go”版本节点源码除了在“RPC”依赖包的“client.go”源码文件中提供了“Call”函数（方法）外，还提供了一个能够支持发起批量“RPC”请求的函数，即“BatchCall”，如图 4-64 所示。

```go
// BatchCall sends all given requests as a single batch and waits for the server
// to return a response for all of them.
//
// In contrast to Call, BatchCall only returns I/O errors. Any error specific to
// a request is reported through the Error field of the corresponding BatchElem.
//
// Note that batch calls may not be executed atomically on the server side.
func (c *Client) BatchCall(b []BatchElem) error {
    ctx := context.Background()
    return c.BatchCallContext(ctx, b)
}
```

图 4-64　client.go 源码文件中的 BatchCall 函数

“BatchCall”函数可以让我们发起批量的“RPC”请求，对于批量的查询，例如交易记录批量查询、批量 ERC20 代币余额查询等，都可以使用这个函数来达到目的。接下来我们认识一下该函数的入参“BatchElem”。

“BatchElem”是一个结构体类型，其内部的构造如图 4-65 所示。

“method”参数代表的意思和“Call”函数中的一样，就是所要请求的以太坊“RPC”接口的名称。“Args”代表的是泛型参数数组，数组内的每个元素一一对应每个单次请求所需的参数。“Result”对应一次返回的结果，类型是泛型，这个值在传入时一般是数组，即用数组来存储返回的结果，数组内的每个元素对应的也是单次请求所产生的结果。最后的一个“Error”是错误类型，

如果该次请求有错误发生，那么将由它来存储和错误有关的内容。

```
66	// BatchElem is an element in a batch request.
67	type BatchElem struct {
68		Method string
69		Args   []interface{}
70		// The result is unmarshaled into this field. Result must be set to a
71		// non-nil pointer value of the desired type, otherwise the response will be
72		// discarded.
73		Result interface{}
74		// Error is set if the server returns an error for this request, or if
75		// unmarshaling into Result fails. It is not set for I/O errors.
76		Error error
77	}
```

图 4-65　BatchElem 结构体在源码中的定义

## 4.9.5　批量获取交易信息

现在我们来使用“BatchCall”函数实现批量获取交易信息的接口。

首先准备构造入参的“BatchElem”参数。对于“method”来说，我们依然使用根据交易哈希值来获取交易信息的接口方法，所以“method”是“eth_getTransactionByHash”。

因为“eth_getTransactionByHash”是根据每笔交易的哈希值来查询的，所以在批量查询的情况下，“Args”参数对应的就是哈希值字符串的数组。同样地，此时的“Result”对应的也应该是交易信息结构体“Transaction”数组的指针。

根据上面的分析，编写如下代码，函数名称是“GetTransactions”。

```
// 根据交易哈希值字符串的数组批量获取对应的交易信息
func (r *ETHRPCRequester) GetTransactions(txHashs []string)
([]*model.Transaction,error) {
   name := "eth_getTransactionByHash"
   // 结果数组存储的是每个请求的结果指针，也就是引用
   rets := []*model.Transaction{}
   // 获取哈希值数组的长度，方便在循环中逐个实例化 BatchElem
   size := len(txHashs)

   reqs := []rpc.BatchElem{}
   for i:=0;i<size;i++ {
      ret := model.Transaction{}
      // 实例化每个 BatchElem
      req := rpc.BatchElem{
         Method:name,
         Args:[]interface{}{txHashs[i]},
         // &ret 传入单个请求的结果引用，保证它在函数内部被修改值后，回到函数外时仍然有效
         Result: &ret,
      }
      reqs = append(reqs,req)  // 将每个 BatchElem 添加到 BatchElem 数组
      rets = append(rets,&ret) // 每个请求的结果引用添加到结果数组中
   }
   err := r.client.GetRpc().BatchCall(reqs) // 传入 BatchElem 数组，发起批量请求
   return rets,err
}
```

编写单元测试代码。这里我们故意制造一笔不存在的交易，观察查询后返回的结果是什么，

有关合法交易哈希值的获取，请参考“实现获取交易信息”一节，测试代码如下：

```
func Test_GetTransactions(t *testing.T) {
   nodeUrl  := https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b
   txHash_1:= "0x53c5b03e392d6aa68a0df26b6d466ae8fbd1c2c5b74f9baae05434ec9a18a282"
   txHash_2:= "0x53c5b03e392d6aa68a0df26b6d466ae8fbd1c2c5b74f9baae05434ec9a18a281"
   txHash_3:= "0x711ddd5f223f970aa0ebc32304a880a8c2ec45ee134b4f41dd4da264f72e1afc"

   // txHash_1 是存在的，_2 是伪造的，_3 也是存在的
   txHashs  := []string{}
   txHashs = append(txHashs,txHash_1,txHash_2,txHash_3)

   if txHashs == nil  || len(txHashs)  == 0 {
      // 这里演示在调用 RPC 接口函数的时候，都要先进行入参的合法性判断
      fmt.Println("非法的交易哈希值数组")
      return
   }
   txInfos,err  := NewETHRPCRequester(nodeUrl).GetTransactions(txHashs)
   if err != nil {
      // 查询失败，打印出信息
      fmt.Println("查询交易失败，信息是：",err.Error())
      return
   }
   // 查询成功，将 transaction 结果的结构体先以 json 格式序列化，再以 string 格式输出
   json,_  := json.Marshal(txInfos)
   fmt.Println(string(json))
}
```

最终返回的结果如图 4-66 所示，我们可以发现，对于不存在的交易查询后的结果信息是，其每个内部的字段值都是空字符串。根据这个特点，我们就能通过判断查询后的结果结构体中的“hash”字段值是不是空字符串来得出对应的交易是否存在。

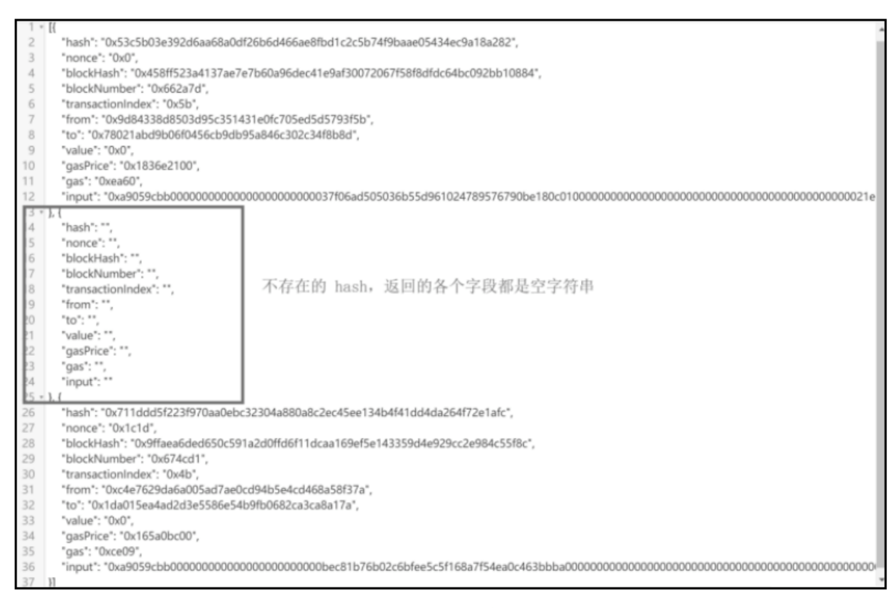



图 4-66　Test_GetTransactions 单元测试函数返回的数据

## 4.9.6 批量获取代币余额

相对于交易信息的查询，对钱包地址所拥有的链上“Token”资产余额数量的查询更为重要，这个功能几乎在所有的钱包和交易所应用中都会用到。这里的“Token”现在已被广泛认为是代币，“Token”的余额也就被称为是代币的余额了。

本节我们依然使用“BatchCall”函数来实现一个专门用来根据用户的以太坊钱包地址和代币地址进行批量代币余额查询的接口。

首先回顾前面一节，在以太坊中，“Token”是一个统称，基于不同智能合约下所发布的“Token”还可以细分为 ERC20 类“Token”、ERC721 类“Token”等，但是为了方便记忆，我们统称为代币。

在目前的代币中，主要分两大类，这两大类代币余额的获取在以太坊节点中所对应的“RPC”接口并不相同，它们分别是：

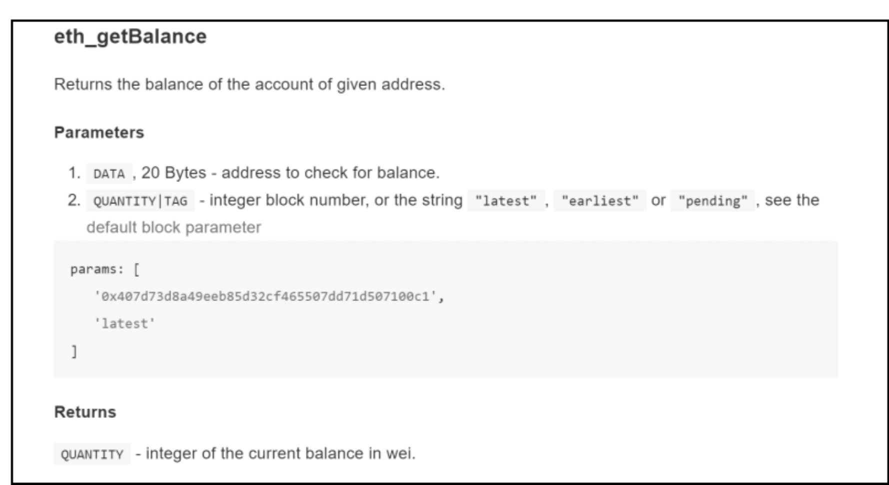
（1）ETH（以太坊），非智能合约类代币，获取 ETH 余额使用的接口名称是“eth_getBalance”。

（2）智能合约类代币，例如 CDCC，获取合约类代币余额的接口名称是“eth_call”，这个接口的“data”参数“methodId”部分取值为“balanceOf”的哈希规则值。

上面所提到的两个接口包括其参数的详细介绍在“重要接口的含义详解”和“交易参数的说明”中已经讲过，此处不再赘述。

### 1. 获取 ETH 余额

要编写获取 ETH 余额的代码，可按照获取交易信息接口的编写步骤进行，首先需要确认好“methodName”参数、“Args”入参和返回结果的结构字段，对应 ETH 余额的“eth_getBalance”接口，可以直接查询接口文档获得所需信息。如图 4-67 所示，“Args”的参数有两个：第一个是所要查询的以太坊地址，即要查询谁的余额；第二个是区块号参数，取值范围是 latest、earliest 或 pending，这 3 个参数的含义在“重要接口的含义详解”一节中已经详细讲解过，关于余额的获取，恒定取值为“latest”即可。



**eth_getBalance**

Returns the balance of the account of given address.

**Parameters**

1. `DATA`, 20 Bytes - address to check for balance.
2. `QUANTITY|TAG` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

```
params: [
   '0x407d73d8a49eeb85d32cf465507dd71d507100c1',
   'latest'
]
```

**Returns**

`QUANTITY` - integer of the current balance in wei.

图 4-67　eth_getBalance 在文档中的介绍

以太坊 ETH 余额获取的相关函数如下所示：

```
// 单笔查询：根据以太坊地址，查询以太坊 eth 的余额
func (r *ETHRPCRequester) GetETHBalance(address string) (string,error) {
    name := "eth_getBalance"
    result := ""
    // 对应文档，第一个参数就是要被查询的以太坊地址，第二个参数就是 latest
    err := r.client.GetRpc().Call(&result,name,address,"latest")
    if err != nil {
        return "",err
    }
    if result == "" {
        return "",errors.New("eth balance is null")
    }
    // 因为查询所返回的结果是一个十进制的字符串，
    // 为了方便阅读，我们在下面使用 go 的大数处理将其转换为十进制数，
    // 并防止数位溢出
    ten,_ := new(big.Int).SetString(result[2:],16)
    return ten.String(),nil
}

// 批量查询：根据以太坊地址数组，查询以太坊 eth 的余额
func (r *ETHRPCRequester) GetETHBalances(addresss []string) ([]string,error) {
    name := "eth_getBalance"
    // 结果数组存储的是每个请求的结果指针，也就是引用
    rets := []*string{}
    // 获取 addresss 数组的长度，方便在循环中逐个实例化 BatchElem
    size := len(addresss)
    reqs := []rpc.BatchElem{}
    for i:=0;i<size;i++ {
        ret := ""
        // 实例化每个 BatchElem
        req := rpc.BatchElem{
            Method:name,
            Args:[]interface{}{addresss[i],"latest"},
            // &ret 传入单个请求的结果引用，保证它在函数内部被修改值后，回到函数外时仍然有效
            Result: &ret,
        }
        reqs = append(reqs,req)  // 将每个 BatchElem 添加到 BatchElem 数组
        rets = append(rets,&ret)  // 每个请求的结果引用添加到结果数组中
    }
    err := r.client.GetRpc().BatchCall(reqs) // 传入 BatchElem 数组，发起批量请求
    if err != nil {
        return nil,err
    }
    // 查询每个请求有没有错误
    for _,req := range reqs {
        if req.Error != nil {
            return nil,req.Error // 返回错误
        }
    }
    finalRet := []string{}
    for _,item := range rets {
        ten,_ := new(big.Int).SetString((*item)[2:],16)
        finalRet = append(finalRet,ten.String())
    }
```

```
    return finalRet,err
}
```

代码最后返回的结果处有一个技术要点，由于查询以太坊“eth_getBalance”接口返回的结果是一个十六进制的字符串，且这个数值的十进制格式是乘上了 ETH 的“decimals”位数幂次方，即这个数的十进制形式是很大的，例如 5 个 ETH，它在函数中最终返回的数值转为十进制时是 $5 \times 10^{18}$，其中 18 就是 ETH 的“decimals”（以太币单位精确到小数点后的位数），$10^{18}$ 就是位数的次方值。

结果是如此大的数值，在这种情况下，已经无法使用“int”类型存储它，因为超过了“int”可表示的最大数的上限。所以在处理以太坊相关的大数值参数或者结果的时候，必须采用大数来存储，或者使用字符串来表示。

同样地，在获取非 ETH 类的代币余额数值的时候，其最终的十进制格式也是乘上合约中所设置的“decimals”位数的幂次方。记住，并不是所有的代币的“decimals”都是同一个值，在前面“实现 ERC20 代币智能合约”一节中已经介绍过“decimals”值是可以在合约代码中进行自定义设置的。

在编写好函数之后，按照惯例编写它们各自对应的单元测试代码，进行单元测试，将所查询出的 ETH 值和区块链浏览器中所查询出的值进行对比，即可验证结果。单元测试的代码如下，依然是编写在我们前面所创建的“ethrpc_test.go”单元测试文件中。

```
// 单笔交易的单元测试函数
func Test_GetETHBalance(t *testing.T) {
    nodeUrl := "https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b"
    address := "0x0D0707963952f2fBA59dD06f2b425ace40b492Fe"
    if address == "" || len(address) != 42 {
        // 这里演示在调用 rpc 接口函数的时候，都要先进行入参的合法性判断
        fmt.Println("非法的交易地址值")
        return
    }
    balance,err := NewETHRPCRequester(nodeUrl).GetETHBalance(address)
    if err != nil {
        // 查询失败，打印出信息
        fmt.Println("查询 eth 余额失败，信息是：",err.Error())
        return
    }
    fmt.Println(balance)
}

// 批量交易的单元测试函数
func Test_GetETHBalances(t *testing.T) {
    nodeUrl := "https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b"

    address1 := "0x0D0707963952f2fBA59dD06f2b425ace40b492Fe"  // 第一个地址
    address2 := "0xf89260db97765A00a343aba8e5682715804769ca"  // 第二个地址

    address := []string{address1,address2}

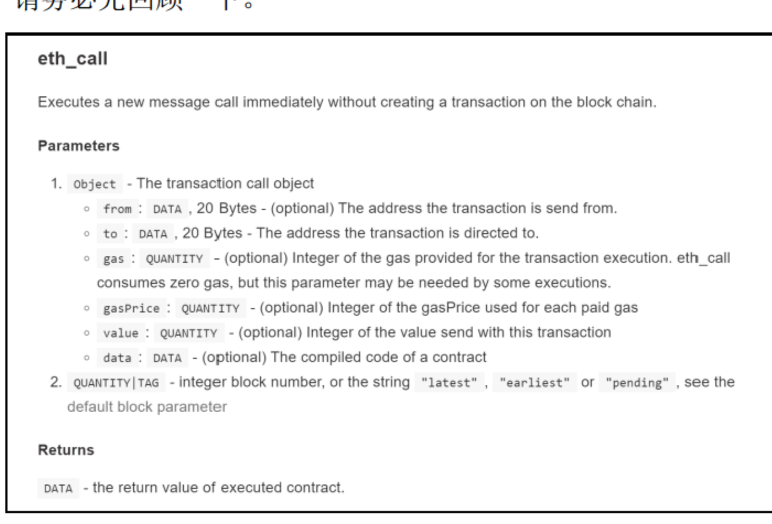
    balance,err := NewETHRPCRequester(nodeUrl).GetETHBalances(address)
    if err != nil {
        // 查询失败，打印出信息
```

```
        fmt.Println("查询 eth 余额失败，信息是：",err.Error())
        return
    }
    fmt.Println(balance)
}
```

### 2. “eth_call”获取合约类代币余额

以太坊的“eth_call”是一个功能非常丰富的接口，在前面的章节中曾多次提到过它，并在“重要接口的含义详解”一节中对它进行过详细的介绍。本节要介绍的获取合约类代币余额的函数也是通过访问该接口来实现的。

首先确定“eth_call”接口的入参。继续查询文档，如图 4-68 所示。“Args”对应的参数共 7 个，每个参数的含义请参考“交易参数的说明”一节，其中第 7 个参数是区块号，对应 3 种取值选择，如果忘记了，请务必先回顾一下。



**eth_call**

Executes a new message call immediately without creating a transaction on the block chain.

**Parameters**

1. `Object` - The transaction call object
    - `from`: `DATA`, 20 Bytes - (optional) The address the transaction is send from.
    - `to`: `DATA`, 20 Bytes - The address the transaction is directed to.
    - `gas`: `QUANTITY` - (optional) Integer of the gas provided for the transaction execution. eth_call consumes zero gas, but this parameter may be needed by some executions.
    - `gasPrice`: `QUANTITY` - (optional) Integer of the gasPrice used for each paid gas
    - `value`: `QUANTITY` - (optional) Integer of the value send with this transaction
    - `data`: `DATA` - (optional) The compiled code of a contract
2. `QUANTITY|TAG` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

**Returns**

`DATA` - the return value of executed contract.

图 4-68 eth_call 函数在文档中的介绍

注意，“eth_call”接口在查询代币余额时，“data”参数中的“methodId”必须根据当前代币对应智能合约的余额查询函数来定，并没有一个固定的值。

本例我们选定基于 ERC20 标准的智能合约余额函数“balanceOf”来演示在“eth_call”中通过设置“balanceOf”的“methodId”来达到访问代币余额的目的。

根据“交易参数的说明”一节中所谈到的，“ERC20”标准余额查询函数“balanceOf”所对应的“methodId”值就是“0x70a08231”，此时“data”的第一个参数就是所要查询余额对应的以太坊地址，参数的格式也在“交易参数的说明”一节做过详细介绍。

相对于 ETH 余额获取函数，合约类代币余额的获取函数相对来说要复杂一些，因为它除了需要被查询地址参数之外，还需要合约的以太坊地址及当前合约所对应代币的“decimals”位数的值。

首先定义好“eth_call”接口所需的参数结构，用来当作以后调用“eth_call”接口的参数结构体，根据上述文档的提示，我们在项目中新建一个“eth_call_arg.go”文件，用来放置该结构体，如图 4-69 所示。

图 4-69　新建 eth_call_arg.go 文件

代码中的“common.Address”数据类型是以太坊依赖包的地址类型，其原型是“[20]byte”数组。

```
// 以太坊 eth_call 的参数结构体
type CallArg struct {
   // common.Address 是以太坊依赖包的地址类型，其原型是 [20]byte 数组
   From     common.Address `json:"from"`
   To       common.Address `json:"to"`
   Gas      string `json:"gas"`
   GasPrice string `json:"gas_price"`
   Value    string `json:"value"`
   Data     string `json:"data"`     // 这个就是 data
   Nonce    string `json:"nonce"`
}
```

“ERC20”代币余额批量获取的函数如下所示：

```
// ERC20BalanceRpcReq 是查询 ERC20 代币的参数集合结构体
type ERC20BalanceRpcReq struct {
   ContractAddress string   // 合约的以太坊地址
   UserAddress     string   // 用户的以太坊地址
   ContractDecimal int      // 合约所对应代币单位精确到小数点后的位数
}
// 批量查询：根据以太坊地址数组，查询 ERC20 代币的余额
func (r *ETHRPCRequester) GetERC20Balances(paramArr []ERC20BalanceRpcReq)
([]string,error) {
   name     := "eth_call"
   methodId := "0x70a08231" // 这个就是 balanceOf 的 methodId
   // 结果数组存储的是每个请求的结果指针，也就是引用
   rets := []*string{}
   // 获取参数数组的长度，方便在循环中逐个实例化 BatchElem
   size := len(paramArr)
   reqs := []rpc.BatchElem{}
   for i:=0;i<size;i++ {
      ret := ""
      arg := &model.CallArg{}
      userAddress := paramArr[i].UserAddress
      // 下面是针对访问 balanceOf 时的必需参数，查询余额是不需要燃料费的，所有不需要设置 Gas
      arg.To   = common.HexToAddress(paramArr[i].ContractAddress)
      // data 参数的组合格式见 “交易参数的说明” 一节中的详解
      arg.Data = methodId+"000000000000000000000000"+userAddress[2:]
      // 实例化每个 BatchElem
```

```
        req := rpc.BatchElem{
            Method:name,
            Args:[]interface{}{arg,"latest"},
            // &ret 传入单个请求的结果引用，这样做是为保证它在函数内部被修改值后，
            //回到函数外部时值仍有效
                Result: &ret,
        }
        reqs = append(reqs,req)   // 将每个 BatchElem 添加到 BatchElem 数组
        rets = append(rets,&ret)  // 每个请求的结果引用添加到结果数组中
    }
    err := r.client.GetRpc().BatchCall(reqs) // 传入 BatchElem 数组，发起批量请求
    if err != nil {
        return nil,err
    }
    // 查询每个请求有没有错误
    for _,req := range reqs {
        if req.Error != nil {
            return nil,req.Error // 返回错误
        }
    }
    finalRet := []string{}
    for _,item := range rets {
        if *item == "" {
            continue
        }
        ten,_ := new(big.Int).SetString((*item)[2:],16)
        finalRet = append(finalRet,ten.String())
    }
    return finalRet,err
}
```

其中，“ERC20BalanceRpcReq”是封装的请求结构体，因为批量获取“ERC20”代币的函数的入参达到了 3 个，为了方便管理和保持代码的可读性，所以将这些参数都放进一个结构体内，各行代码的含义见注释。在“data”参数行，“000000000000000000000000”和“userAddress[2:]”的拼接组成了“交易参数的说明”一节中所讲到的 64 个字符的参数，前面的 20 个零字符加上去掉了“0x”字符的“userAddress”地址，共 64 个字符。

单元测试代码及其运行结果如下：

```
// 单元测试：批量获取代币值
func Test_GetERC20Balances(t *testing.T) {
    nodeUrl := "https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b"

    address  := "0xc58AD8Ff428c354bb849d1dCf1EDCcAC3F102C8E"  // 钱包地址
    contract1 := "0x78021ABD9b06f0456cB9DB95a846C302c34f8b8D"  // 合约地址 1
    contract2 := "0xB8c77482e45F1F44dE1745F52C74426C631bDD52"  // 合约地址 2

    params := []ERC20BalanceRpcReq{}
    item := ERC20BalanceRpcReq{}
    item.ContractAddress = contract1
    item.UserAddress = address
    item.ContractDecimal = 18

    params = append(params,item)
```

```
    item.ContractAddress = contract2
    params = append(params,item)

    balance,err := NewETHRPCRequester(nodeUrl).GetERC20Balances(params)
    if err != nil {
        // 查询失败，打印出信息
        fmt.Println("查询 eth 余额失败，信息是：",err.Error())
        return
    }
    fmt.Println(balance)
}
```

查询结果如图 4-70 所示。




图 4-70　Test_GetERC20Balances 单元测试函数获取 ERC20 代币余额的返回结果

以上只是使用“eth_call”接口的例子之一，在实际的开发中，很多智能合约函数的调用都是可以通过访问该接口来达到目的的。

## 4.9.7　获取最新区块号

以太坊区块链上的区块是在不断地被生成的，区块中包含了数据。以太坊中继对区块进行遍历时，需要获取每个区块中的数据，需要在生成的区块胜出之后立刻获取到它的区块号，然后使用获取区块信息相关的接口函数根据区块号来获取到它的内部信息。

以太坊提供的获取链上最新区块号的接口名称是“eth_blockNumber”，该接口不需要参数，返回的区块号结果也是一个十六进制的字符串。通过查询接口文档，我们可以看到它的传参及返回结果，如图 4-71 所示。

依然在“ETH_RPC_Requester.go”文件中编写请求代码，我们设置最终返回的结果是一个大整数类型“big.Int”。

```
// 获取以太坊最新生成区块的区块号
func (r *ETHRPCRequester) GetLatestBlockNumber() (*big.Int, error) {
    methodName := "eth_blockNumber"
    number := ""  // 存储结果
    err := r.client.client.Call(&number, methodName) // eth_blockNumber 不需要参数
    if err != nil {
        return nil, fmt.Errorf("获取最新区块号失败! %s", err.Error())
```

```
    }
    ten,_ := new(big.Int).SetString(number[2:],16) // 十六进制转为十进制大整数
    return ten, nil
}
```

**eth_blockNumber**

Returns the number of most recent block.

**Parameters**

none    none 代表不需要参数

**Returns**

`QUANTITY` - integer of the current block number the client is on.

**Example**

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_blockNumber","params":[],"id":83}'

// Result
{
  "id":83,
  "jsonrpc": "2.0",
  "result": "0x4b7" // 1207
}
```

图 4-71　eth_blockNumber 在文档中的介绍

单元测试代码如下：

```
// 单元测试：获取以太坊最新生成区块的区块号
func TestGetLatestBlockNumber(t *testing.T) {
    nodeUrl := "https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b"
    number, err := NewETHRPCRequester(nodeUrl).GetLatestBlockNumber()
    if err != nil {
        // 查询失败，打印出信息
        fmt.Println("获取区块号失败，信息是：", err.Error())
        return
    }
    fmt.Println("10 进制：", number.String())
}
```

运行后可知，已经获取最新的区块号，这个区块号就是当前刚刚连接上以太坊公链的区块的号码，如图 4-72 所示。

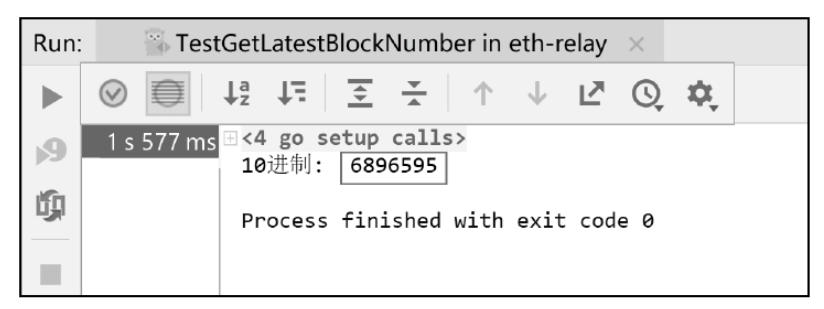



图 4-72　TestGetLatestBlockNumber 单元测试函数的运行结果

## 4.9.8　根据区块号获取区块信息

对应“获取最新区块号”一节中获取的区块号，本节我们根据区块号来获取区块的数据信息。

以太坊提供的接口名称是“eth_getBlockByNumber”，通过查询接口文档，我们可以看到它的传参及返回结果，如图 4-73 所示。

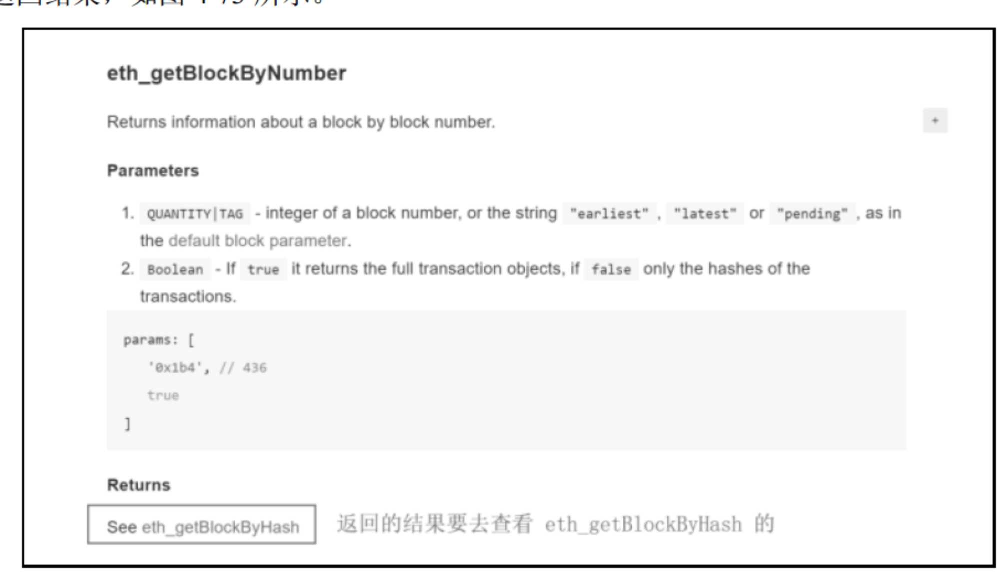



图 4-73　eth_getBlockByNumber 在文档中的介绍

其中，返回结果和另外一个根据区块哈希值来获取区块信息的接口是一样的，而参数则需要两个。第一个参数是用十六进制字符串表示的区块号，刚好对应获取最新区块号接口返回的结果。第二个参数是个布尔值，它的取值可能有：

- true，返回区块中所有被打包进去的交易的完整信息数组。
- false，返回区块中所有被打包进去的交易的哈希值数组。

在接口文档中查询“eth_getBlockByHash”接口的返回结果，我们可以看到能够获取的区块信息，如图 4-74 所示。

信息是比较丰富的，其中的大部分字段，我们在“区块的组成”一节中都做过详细介绍，其中带“root”的字段是默克尔树的根部的“哈希”值，而“transactions”就是被打包进区块中的交易信息数组。

对于上述各个字段，我们需要在代码中定义一个结构体，用来对应的字段来存储它们。在项目的“model”文件夹下创建“full_block.go”文件，如图 4-75 所示。

**Returns**

`Object` - A block object, or `null` when no block was found:

- `number`: `QUANTITY` - the block number. `null` when its pending block.
- `hash`: `DATA`, 32 Bytes - hash of the block. `null` when its pending block.
- `parentHash`: `DATA`, 32 Bytes - hash of the parent block.
- `nonce`: `DATA`, 8 Bytes - hash of the generated proof-of-work. `null` when its pending block.
- `sha3Uncles`: `DATA`, 32 Bytes - SHA3 of the uncles data in the block.
- `logsBloom`: `DATA`, 256 Bytes - the bloom filter for the logs of the block. `null` when its pending block.
- `transactionsRoot`: `DATA`, 32 Bytes - the root of the transaction trie of the block.
- `stateRoot`: `DATA`, 32 Bytes - the root of the final state trie of the block.
- `receiptsRoot`: `DATA`, 32 Bytes - the root of the receipts trie of the block.
- `miner`: `DATA`, 20 Bytes - the address of the beneficiary to whom the mining rewards were given.
- `difficulty`: `QUANTITY` - integer of the difficulty for this block.
- `totalDifficulty`: `QUANTITY` - integer of the total difficulty of the chain until this block.
- `extraData`: `DATA` - the "extra data" field of this block.
- `size`: `QUANTITY` - integer the size of this block in bytes.
- `gasLimit`: `QUANTITY` - the maximum gas allowed in this block.
- `gasUsed`: `QUANTITY` - the total used gas by all transactions in this block.
- `timestamp`: `QUANTITY` - the unix timestamp for when the block was collated.
- `transactions`: `Array` - Array of transaction objects, or 32 Bytes transaction hashes depending on the last given parameter.
- `uncles`: `Array` - Array of uncle hashes.

图 4-74　eth_getBlockByHash 函数的返回值及其介绍

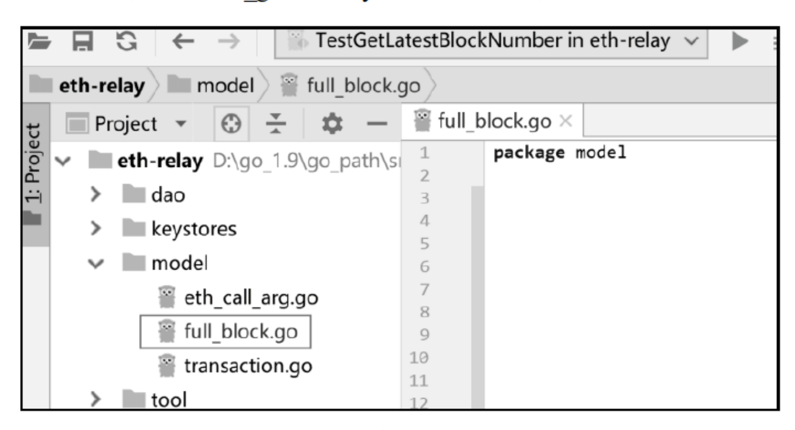



图 4-75　创建 full_block 文件

结构体代码如下所示，其中每个字段都添加了注释。

```
// 根据文档定义出区块信息的结构体
type FullBlock struct {
  Number    string   `json:"number"`   // 区块号
  Hash    string   `json:"hash"`   // 区块的哈希值
  ParentHash     string   `json:"parentHash"`   // 父区块的哈希值
  Nonce    string   `json:"nonce"`   // 区块的序列号
  Sha3Uncles     string   `json:"sha3Uncles"`   // 当前区块如果打包了叔块,
// 那么它是叔块的 sha3 加密值
  LogsBloom    string    `json:"logsBloom"`   // 当前区块的布隆过滤器日志
  TransactionsRoot      string    `json:"transactionsRoot"`   // 交易默克尔树的根部
hash 值
  ReceiptsRoot    string    `json:"stateRoot"`   // 收据默克尔树的根部的哈希值
  Miner    string    `json:"miner"`   // 挖出此区块的矿工的以太坊地址值
```

```
    Difficulty  string  `json:"difficulty"`  // 这个区块的难度值
    TotalDifficulty   string  `json:"totalDifficulty"`  // 这个块的链的总难度
    ExtraData  string  `json:"extraData"`  // 区块的附属数据
    Size  string  `json:"size"`  // 这个区块总数据量的大小
    GasLimit  string  `json:"gasLimit"`   // 区块的 GasLimit，注意它和交易的不一样
    GasUsed  string  `json:"gasUsed"`  // 当前该区块已经打包了的交易的总燃料费
    Timestamp  string  `json:"timestamp"`  // 区块被确认核实的时间戳，单位为秒
    Uncles  []string  `json:"uncles"`  // 叔块的哈希数组
    Transactions  []interface{}  `json:"transactions"`// 所有被打包了的交易的数组
}
```

最终根据区块号获取区块信息的接口请求函数如下所示，返回的结果就是上面所定义的“FullBlock”区块结构体。

```
// 根据区块号获取区块信息
func (r *ETHRPCRequester) GetBlockInfoByNumber(blockNumber *big.Int)
(*model.FullBlock, error) {
    number := fmt.Sprintf("%#x", blockNumber)  // 将 big.Int 转为十六进制字符串
    methodName := "eth_getBlockByNumber"
    fullBlock := model.FullBlock{}  // 区块信息结构体
    // eth_getBlockByNumber 的第二个参数:
    // 若是 true，则返回完整的区块信息；若是 false，则 transaction 部分只返回交易哈希数组
    err := r.client.client.Call(&fullBlock, methodName, number, true)
    if err != nil {
        return nil, fmt.Errorf("get block info failed! %s", err.Error())
    }
    if fullBlock.Number == "" {
        return nil, fmt.Errorf("block info is empty %s",blockNumber.String())
    }
    return &fullBlock, nil
}
```

单元测试部分需要结合获取最新区块号的接口进行，代码如下：

```
// 单元测试：根据区块号获取区块信息
func TestGetFullBlockInfo(t *testing.T) {
    nodeUrl := "https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b"
    requester := NewETHRPCRequester(nodeUrl)
    number, _ := requester.GetLatestBlockNumber() // 获取区块号
    fmt.Println("区块号是:\n",number)
    fullBlock, err := requester.GetBlockInfoByNumber(number) // 获取区块信息
    if err != nil {
        // 查询失败，打印出信息
        fmt.Println("获取区块信息失败，信息是：", err.Error())
        return
    }
    // 查询成功，将区块结果的结构体先以 json 格式序列化，再以 string 格式输出
    json1, _ := json.Marshal(fullBlock)
    fmt.Println("根据区块号获取区块信息:\n", string(json1))
}
```

运行后，我们可以看到成功获取区块号后输出的区块“json”数据，数据量是比较多的，且都是十六进制的格式。如图 4-76 所示。





```
{
  "number": "0x693c84",
  "hash": "0xda4788b39b9fdb5f8b18c9820ef5c16a3fe4bfc621d9208d4676fcf1e75b40a9",
  "parentHash": "0x2927c36f218d2650e1a3cd16084908749992df2fc16b1dbaaca25fcc6cc3913d",
  "nonce": "0x4fc7c6dc1b5bb32f",
  "sha3Uncles": "0x29dc24b478766d55d62eb246cbc48c5287a2f4acaa20ce37212933f613668472",
  "logsBloom": "0x10061880025050040828 1a1c008e8086921212254c002008805801201e4330d0481a0c002820
  "transactionsRoot": "0xcc6d5a1a35ee7c8a90312e9e492948774ba168a045b0474d723dd8a5bd9b906b",
  "stateRoot": "0x7cba18ebd99c8274c768f97d39d05984c41264d0c48fcb9d3968a4bdd04e7058",
  "miner": "0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c",
  "difficulty": "0x7d8b4be4861ee",
  "totalDifficulty": "0x1c200faa01744ee1bcf",
  "extraData": "0x737061726b706f6f6c2d6574682d636e2d687a",
  "size": "0x31db",
  "gasLimit": "0x7a121d",
  "gasUsed": "0x79d9c1",
  "timestamp": "0x5c162d13",
  "uncles": ["0x072b9111c173f05316e2d65f459050f1846c24cd8f427ac220e995698670e0da"],
  "transactions": [{
    "blockHash": "0xda4788b39b9fdb5f8b18c9820ef5c16a3fe4bfc621d9208d4676fcf1e75b40a9",
    "blockNumber": "0x693c84",
    "from": "0xd1c2da7eaa1f07a034325f54c1c7175523d0f2e1",
    "gas": "0x15f90",
    "gasPrice": "0x3b9aca000",
    "hash": "0xf4e2446c2bad2735d0dd1c70d10ae44d3fe12e15401a18e34d8de4b4013e8479",
    "input": "0x",
    "nonce": "0xaf1",
    "r": "0x51f5ddc1d38e3ecb714c76333956d65309351b7c897098c7437a60acba760859",
    "s": "0x5fc41abcf2e5cc298d544eb524c95d1ae54e2183648084feeba51ce61899b4ea",
    "to": "0x22c6d93a2020fcd133963b54bff4958a0ffd6752",
    "transactionIndex": "0x0",
    "v": "0x25",
    "value": "0xde0b6b3a7640000"
  }, {
```


图 4-76　TestGetFullBlockInfo 单元测试函数获取的区块信息

## 4.9.9　根据区块哈希值获取区块信息

除了可以根据区块号获取区块的数据外，以太坊还提供了一个“eth_getBlockByHash”接口，该接口可根据区块的“hash”值获取区块的信息。

它和上一节所介绍的根据区块号获取区块信息的“eth_getBlockByNumber”接口除了第一个参数不一样之外，其他的调用参数和返回结果是一模一样的。在学习“eth_getBlockByHash”接口的实现之前，请务必先掌握“根据区块号获取区块信息”一节的内容，即“eth_getBlockByNumber”接口的实现。

如图 4-77 的文档所示，“eth_getBlockByHash”的第一个参数是区块的“hash”值。

**eth_getBlockByHash**

Returns information about a block by hash.

**Parameters**

1. `DATA` , 32 Bytes - Hash of a block.
2. `Boolean` - If `true` it returns the full transaction objects, if `false` only the hashes of the transactions.

```
params: [
  '0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331',
  true
]
```

图 4-77　eth_getBlockByHash 函数在文档中的介绍

接口请求的代码如下：

```
// 根据区块哈希值获取区块信息
func (r *ETHRPCRequester) GetBlockInfoByHash(blockHash string) (*model.FullBlock,
error) {
   methodName := "eth_getBlockByHash"
   fullBlock := model.FullBlock{} // 区块信息结构体
   // eth_getBlockByHash 的第二个参数:
   // 若是 true，则返回完整的区块信息，若是 false，则 transaction 部分只返回交易哈希值数组
   err := r.client.client.Call(&fullBlock, methodName, blockHash, true)
   if err != nil {
      return nil, fmt.Errorf("get block info failed! %s", err.Error())
   }
   if fullBlock.Number == "" {
      return nil, fmt.Errorf("block info is empty %s",blockHash)
   }
   return &fullBlock, nil
}
```

单元测试代码如下，所查询的区块的“hash”（哈希值）对应“根据区块号获取区块信息”一节中所获取的区块的“hash”（哈希值）：

```
// 单元测试：根据区块哈希获取区块信息
func TestGetFullBlockByBlockHash(t *testing.T) {
   nodeUrl := "https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b"
   requester := NewETHRPCRequester(nodeUrl)
   blockHash :=
"0xda4788b39b9fdb5f8b18c9820ef5c16a3fe4bfc621d9208d4676fcf1e75b40a9"
   // 根据区块哈希获取区块信息
   fullBlock, err := requester.GetBlockInfoByHash(blockHash)
   if err != nil {
      // 查询失败，打印出信息
      fmt.Println("获取区块信息失败，信息是：", err.Error())
      return
   }
   json2, _ := json.Marshal(fullBlock)
   fmt.Println("根据区块哈希值获取区块信息\n", string(json2))
}
```

运行结果如图 4-78 所示，对应于区块号“6896772”的信息。

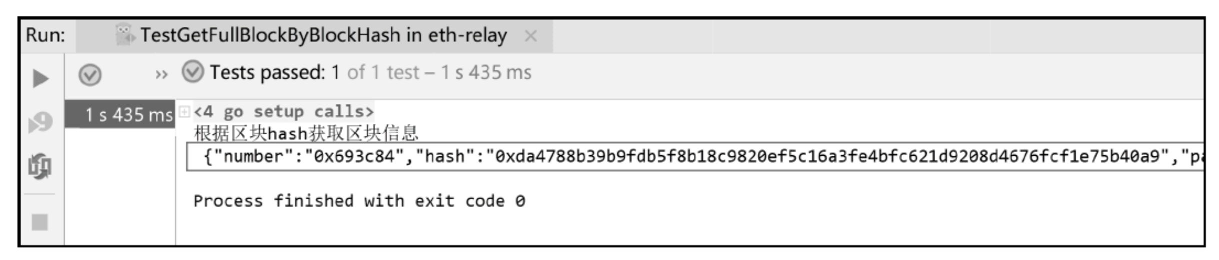



图 4-78　区块号“6896772”的信息

## 4.9.10　使用“eth_call”访问智能合约函数

在“eth_call 获取合约类代币余额”一节中我们已经学习了如何使用“eth_call”，以及运用它来调用智能合约的“balanceOf”函数。在这一节中，我们再使用它来调用在第 3 章中所发布的加法运算智能合约中的加法函数。

调用智能合约中的函数，先要找到以下必需的信息：

（1）被调用的智能合约的以太坊地址。

（2）智能合约中被访问函数的“methodId”。

（3）被访问函数的参数。

由“实现加法程序”一节的内容可知，加法智能合约中两数相加的函数定义如下：

```
function add(uint8 arg1,uint8 arg2) public pure returns (uint8)
```

函数名称是“add”，接收两个无符号 8 位整型参数，最终返回的结果也是无符号 8 位整型数据。

此外加法智能合约发布在以太坊主网上的地址是：

```
0x339dbB357E3BD3c349a912ac3a5A6D4079216911
```

### 1.生成“methodId”

要使用“eth_call”来访问智能合约的函数，必须根据函数的名称生成对应的“methodId”。

“methodId”的生成算法比较复杂，一般我们不需要自己编写核心代码来生成，而是直接使用以太坊源码中提供的函数来生成。

这个封装了“methodId”生成函数的源码文件（见图书 4-79）是：

```
go_path/src/github.com/ethereum/gp-ethereum/accounts/abi/abi.go
```

可以使用“ABI”结构体中的“Methods”成员变量来选出对应的“Method”对象，然后调用“Method”对象内的“Id()”函数来生成“methodId”。如图 4-80 所示，“Id()”就是“Method”对象所在的“method.go”代码文件中的一个函数。

图 4-79　以太坊 Go 版本源码中的 abi.go 代码文件

图 4-80　以太坊 Go 版本源码中提供了生成 methodId 的函数

因为生成“methodId”的是一个协助函数，所以我们把它的代码也编写到“tool”文件夹下的“wallet.go”文件中。

代码如下，在实例化“ABI”结构体对象指针后，使用智能合约的“abi”数据来初始化其内部的变量，其中智能合约的“abi”数据的介绍与提取参见 3.9.2 节和 3.9.3 节的内容。

```
// 根据函数的名称生成 methodId。abiStr 是智能合约的“abi”数据
func MakeMethoId(methodName string,abiStr string) (string,error) {
   abi  := &abi.ABI{}  // 实例化 “ABI” 结构体对象指针
   err := abi.UnmarshalJSON([]byte(abiStr))
   if err != nil {
      return "",err
   }
   // 根据 methodName 获取对应的 Method 对象
   method  := abi.Methods[methodName]
   methodIdBytes  := method.Id()  // 调用生成 methodId 的函数
   methodId  := "0x"+common.Bytes2Hex(methodIdBytes)
   return methodId,nil
}
```

加法智能合约的“abi”数据是：

```
   [ { "constant": true, "inputs": [ { "name": "arg1", "type": "uint8" }, { "name":
```

```
"arg2", "type": "uint8" } ], "name": "add", "outputs": [ { "name": "", "type":
"uint8" } ], "payable": false, "stateMutability": "pure", "type": "function" } ]
```

单元测试代码编写在“tool_test.go”文件中，如下所示：

```
// 单元测试：生成 methodId
func Test_MakeMethodId(t *testing.T) {
   contractABI := // 加法智能合约的 abi 数据
      `[ { "constant": true, "inputs": [ { "name": "arg1", "type": "uint8" },
      { "name": "arg2", "type": "uint8" } ],
      "name": "add", "outputs": [ { "name": "", "type": "uint8" } ],
      "payable": false, "stateMutability": "pure", "type": "function" } ]`;
   methodName := "add" // 加法函数名称
   methodId,err := MakeMethoId(methodName,contractABI)
   if err != nil {
      fmt.Println("生成 methodId 失败",err.Error())
      return
   }
   fmt.Println("生成 methodId 成功",methodId)
}
```

单元测试运行结果如图 4-81 所示。可以看到，最终函数名称为“add”的“methodId”是“0xbb4e3f4d”。

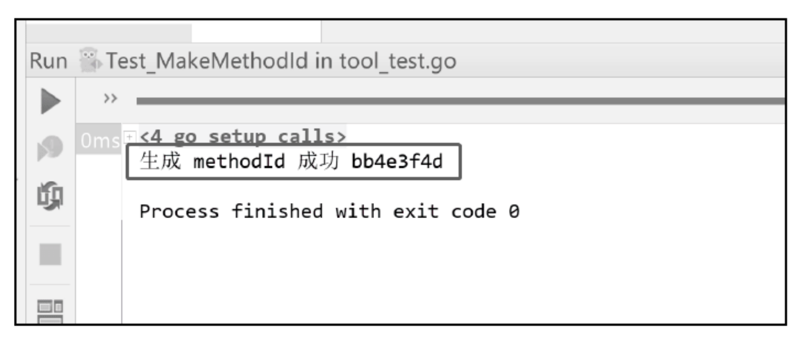



图 4-81　测试运行结果

## 2.访问“add”函数

接下来我们使用以太坊提供的接口“eth_call”来访问合约中的“add”函数，并将结果输出。

在编写访问智能合约的“add”函数之前，首先在以太坊请求者“ETH_RPC_Requester.go”文件中完成一个通用请求以太坊“eth_call”接口的函数，代码如下所示。其中，“model.CallArg”结构体是“eth_call”参数的集合结构体。

```
// 使用 eth_call 调用智能合约的函数
// 第一个参数是接收结果的结构体，第二个参数是 eth_call 参数集合结构体
func (r *ETHRPCRequester) ETHCall(result interface{},arg model.CallArg) error {
   methodName := "eth_call"
   err  := r.client.client.Call(result, methodName, arg,"latest")
   if err != nil {
      return fmt.Errorf("eth_call failed! %s", err.Error())
   }
   return nil
}
```

在“ETHCall”的单元测试里，我们结合上一节的“MakeMethodId”函数来实现访问加法智能合约的“add”函数。

单元测试代码如下：

```
// 单元测试：使用 eth_call 访问智能合约的函数
func Test_ETHCall(t *testing.T) {
   contractABI := // 加法智能合约的 abi 数据
      `[ { "constant": true, "inputs": [ { "name": "arg1", "type": "uint8" },
      { "name": "arg2", "type": "uint8" } ],
      "name": "add", "outputs": [ { "name": "", "type": "uint8" } ],

      "payable": false, "stateMutability": "pure", "type": "function" } ]`;
   methodName := "add"  // 智能合约中的函数名称
   methodId,err := tool.MakeMethoId(methodName,contractABI) // 生成对应的 methodId
   if err != nil {
      panic(err)
   }
   // 下面要进行的运算是：2+3
   arg1 := common.HexToHash("2").String()[2:] //根据 data 中的参数格式，生成第一个参数
   // arg1 = 0000000000000000000000000000000000000000000000000000000000000002
   arg2 := common.HexToHash("3").String()[2:] //根据 data 中的参数格式，生成第二个参数
   // arg2 = 0000000000000000000000000000000000000000000000000000000000000003
   contractAddress := "0x339dbB357E3BD3c349a912ac3a5A6D4079216911" // 智能合约地址
   args := model.CallArg{
      To  : common.HexToAddress(contractAddress), //  此时的 to 对应的是合约的地址，代
表访问该合约
      Data: methodId + arg1 + arg2, // 组合成 data 的完整格式
      // 下面的无关参数可以不进行赋值，让它们使用默认值
      //Gas:"0x0",
      //GasPrice:"0x0",
      //Value:"0x0",
      //Nonce:"0x0",
   }
   result  := ""  // 结果是一个十六进制字符串
   nodeUrl := https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b
   requester := NewETHRPCRequester(nodeUrl)
   err = requester.ETHCall(&result,args) // 进行调用
   if err != nil {
      panic(err)
   }
   ten,_ := new(big.Int).SetString(result[2:],16)  //将十六进制结果转为十进制
   fmt.Println("调用合约两数相加结果是：",ten.String())
}
```

运行结果如图 4-82 所示。可以看到智能合约已经帮我们计算出了“2+3”的结果，即“5”。




图 4-82　调用链上的加法智能合约后返回的结果

以上就是一个使用以太坊的“eth_call”接口访问智能合约函数的例子。它表明了一般的访问流程，无论智能合约中的函数具备什么功能，都能通过上面的操作流程进行访问。需要注意的是，打包进区块的合约函数不能使用“eth_call”来访问触发，因为这种情况属于以太坊交易，请务必使用以太坊的发送交易接口进行访问触发。

# 第 5 章

# 实现以太坊中继——应用

本章将综合运用前面各章内容，使用绝大部分以太坊 RPC 接口请求函数，采用 Go 语言实现在以太坊 DApp 应用中一个很强大的中继服务支持程序，包括钱包、以太坊交易、区块链分叉检测以及分叉区块数据的存储回滚操作等内容。

## 5.1 创建以太坊钱包

以太坊钱包的创建是我们获取并使用属于自己的以太坊地址的唯一途径。

钱包的创建一般多见于基于移动 App 的软件中，也就是钱包 App 软件。这类钱包 App 软件的钱包创建功能都是脱离服务器端的，也就是可以直接离线断网在 App 中进行钱包生成。之所以脱离网络连接进行钱包的生成，主要有两个原因：

（1）钱包的创建在代码层面涉及大量的数学运算，是比较耗时的，不适合在服务器端进行生成。

（2）可以 100%避免在创建钱包的时候被抓包拦截，钱包信息被截取。如图 5-1 所示是钱包由服务器端生成再将信息传递回客户端的一个演示，其间可能会带来风险。

虽然钱包多由 App 生成，但在一些业务场景下也需要由服务器端来帮助用户生成钱包，例如：

- 中心化钱包。为避免小白用户不会操作，让用户一键生成钱包。
- 中心化交易所。用户一个账户对应多币种钱包，这种情况可由服务器端生成并保存私钥等信息。
- 在服务器端生成钱包。可以设计在钱包创建成功时不返回钱包相关的私密信息给客户端，以避免钱包信息被抓包截取的问题。

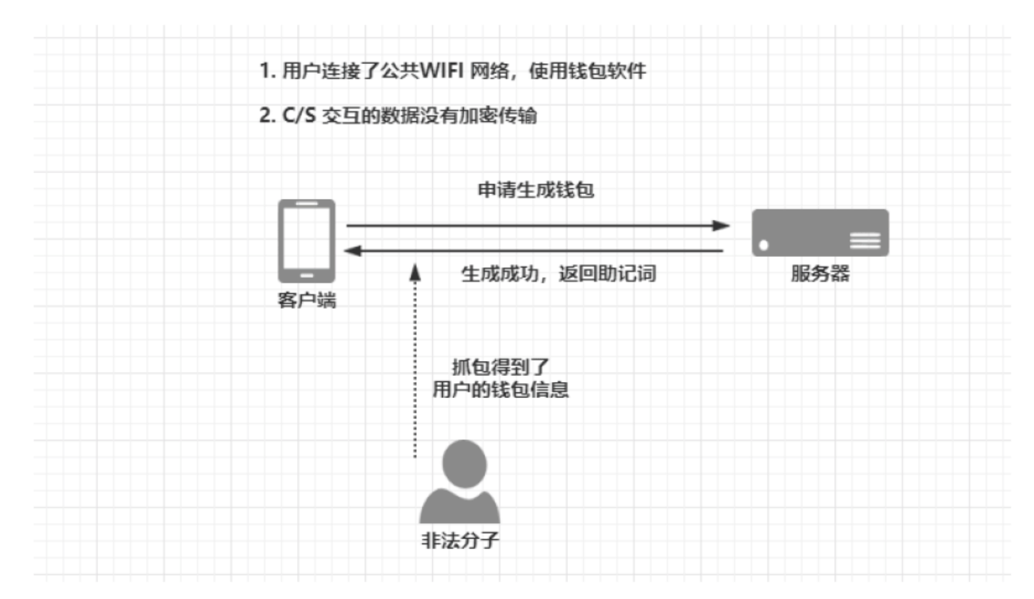



图 5-1　抓包拦截获取钱包信息的演示

## 5.1.1　以太坊钱包术语

在使用代码创建以太坊钱包之前，一定要先理清楚与以太坊钱包相关的术语。在前面的“地址的含义”一节和“Mist”部分的“创建以太坊钱包”一节我们对以太坊的相关术语已经做过一些介绍，这里我们再做一个全面详细的总结。

（1）钱包，是基于交易所的一种客户端钱包管理软件。我们可以用钱包管理自己基于当前这个交易所的虚拟货币资产，钱包里的所有信息（例如公钥、私钥）都基于我们在这个钱包软件上所注册的账号。

（2）账户地址，又称钱包地址，一般不等于公钥，但不排除用户使用的交易所内部的代码将其设置为公钥。一般账户地址由公钥和特定的算法生成，例如由哈希算法生成。账户地址与算法和公钥的关系如下：

算法+公钥=账户地址

（3）密码，由交易所制定，为保护我们的钱包信息而设置的，主要用于加密私钥。密码的作用如下：

- 登录 App，例如交易所推出的某个 App。
- 加密私钥，使之形成 Keystore 文件。

| 注　意 |
| --- |
| 当我们在某些交易所的软件上设置好账号后，对应的登录密码还可能用来加密私钥。 |

（4）公钥，由私钥通过某些算法生成，比如常见的椭圆曲线加密算法。其中，公钥和私钥的关系如下：

- 私钥可以计算出公钥，公钥不能计算出私钥。
- 被公钥锁加密了的数据，只能使用私钥解密。
- 私钥加密数据这个步骤，一般称为数字签名。

（5）私钥，是随机生成的，这个随机数可能有 $2^{256}$ 种。私钥的生成在钱包中的体现是，在我们创建好账号后代码自动生成，并使用我们的账户密码进行加密，形成 KeyStore 文件，保存在手机本地或交易所的数据库中。

私钥、公钥与账户地址的关系如下：

- 算法步骤。首先根据非对称加密算法中的椭圆曲线算法的要求，生成一个随机数作为私钥，再由私钥根据椭圆曲线算法（ECDSA-secp256k1）生成公钥，从公钥的哈希值中提取部分字符串得出账户地址。
- 生成步骤。
  - 随机产生一个私钥。
  - 私钥通过“ECDSA-secp256K1”算法生成公钥，即计算得到私钥在椭圆曲线上对应的公钥。
  - 对公钥做 SHA3 计算，得到一个哈希值，取这个哈希值的后 20 个字节作为账户地址。
- 私钥保存在交易所里，风险很大，如被黑客盗窃，就会造成财产失窃。
- 私钥签名的数据可以用公钥解签。
- 私钥加密数据，我们称之为数字签名。
- 私钥的导出：账户密码+KeyStore 文件+加密算法=私钥。

  某些中心化交易所 App 或钱包 App 不提供导出功能，此时用户便无法知道私钥。

（6）助记词。当我们忘记了所使用的交易所 App 或钱包 App 的登录密码，可用助记词修改密码。理论上，不是所有的交易所 App 或钱包 App 都提供这个功能。

- 助记词一般由 12 或 24 个单词构成，2 个单词之间由 1 个空格隔开，这些单词都来源于一个固定词库（单词表）。
- 组成助记词的单词根据一定的算法挑选得出。
- 助记词实际上就是私钥的另一种表现形式。
- 重要性和私钥一样。
- 助记词能生成私钥，但不能根据私钥推出助记词。

（7）Keystore，一个用来存储私钥数据的被加密了的文件。如果没有对应的加密密码，很难获得该文件的私钥。

- 账户密码+私钥+加密算法=Keystore。
- Keystore 数据使用 Json 格式存储。
- 使用 Keystore 文件恢复私钥时，只需输入加密密码，私钥就能从 Keystore 文件恢复出来。

## 5.1.2 创建钱包

在服务器端创建以太坊钱包，不需要进行以太坊节点接口的访问，可以直接使用以太坊源码提供的依赖库来实现。

如图 5-2 所示，位于“gopath”下的“NewAccount”就是官方提供的可以用来创建以太坊钱包的函数：

github.com/ethereum/go-ethereum/accounts/keystore/keystore.go

图 5-2 以太坊 Go 版源码中的根据密码生成钱包的函数

```
// NewAccount generates a new key and stores it into the key directory,
// encrypting it with the passphrase.
func (ks *KeyStore) NewAccount(passphrase string) (accounts.Account, error) {
   _, account, err := storeNewKey(ks.storage, crand.Reader, passphrase)
   if err != nil {
      return accounts.Account{}, err
   }
   // Add the account to the cache immediately rather
   // than waiting for file system notifications to pick it up.
   ks.cache.add(account)
   ks.refreshWallets()
   return account, nil
}
```

其中，“passphrase”是设置的密码参数，最终会用来结合私钥生成“keystore”文件。创建钱包的步骤是：

（1）根据随机数创建“私钥”。

（2）根据私钥生成“公钥”。

（3）根据公钥生成“钱包地址”。

（4）将“私钥”结合所设置的“密码”生成“keystore”文件，存储起来。

要想使用“NewAccount”函数，首先须实例化一个“KeyStore”对象，因为“NewAccount”函数被定义为“func (ks*KeyStore)”类型，代表它是实例指针的公有函数。

在钱包创建成功后，所生成的“keystore”文件还要被存储起来，所以需要指定一个存储“keystore”文件的文件夹。我们在项目主文件夹下创建一个子文件夹“keystores”，用来存储钱

包的“keystore”文件，如图 5-3 所示。

图 5-3　创建一个用来存储钱包文件的子文件夹“keystores”

创建钱包的函数依然编写在“ETH_RPC_Requester.go”文件中，代码如下：

```
// 创建以太坊钱包
func (r *ETHRPCRequester) CreateETHWallet(password string) (string,error) {
   if password == "" {
      return "",errors.New("password cant empty")
   }
   if len(password) < 6 {
      return "",errors.New("password's len must more than 6 words")
   }
   keydir := "./keystores" // 用来存储所创建的钱包的 keystore 文件的文件夹
   // StandardScryptN 是 Scrypt 加密算法的标准 N 参数
   // StandardScryptP 是 Scrypt 加密算法的标准 P 参数
   ks := keystore.NewKeyStore(keydir, keystore.StandardScryptN,
keystore.StandardScryptP)
   wallet, err := ks.NewAccount(password) // 传入密码，创建钱包
   if err != nil {
      return "0x", err
   }
   return wallet.Address.String(),nil
}
```

单元测试代码如下：

```
// 单元测试：创建以太坊钱包
func Test_CreateETHWallet(t *testing.T) {
   nodeUrl := "https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b"
   address1,err := NewETHRPCRequester(nodeUrl).CreateETHWallet("13456")
// 演示密码太短的错误
   if err != nil {
      fmt.Println("第一次，创建钱包失败",err.Error())
   }else{
      fmt.Println("第一次，创建钱包成功，以太坊地址是：",address1)
   }
   address2,err := NewETHRPCRequester(nodeUrl).CreateETHWallet("13456aa")
// 创建成功
   if err != nil {
```

```
        fmt.Println("第二次，创建钱包失败",err.Error())
    }else {
        fmt.Println("第二次，创建钱包成功，以太坊地址是：", address2)
    }
}
```

运行结果如图 5-4 所示。

```
Run:  Test_CreateETHWallet in eth-relay
Tests passed: 1 of 1 test – 695 ms
695 ms  <4 go setup calls>
第一次，创建钱包失败 password's len must more than 6 words
第二次，创建钱包成功，以太坊地址是： 0x590c3D81B70DdfF32F74E51f14805915a4C0e2eD

Process finished with exit code 0
```

图 5-4　Test_CreateETHWallet 单元测试函数的运行结果

如图 5-4 所示，新创建好的以太坊钱包地址是 0x590c3D81B70DdfF32F74E51f14805915a4C0e2eD。

进入“keystores”文件夹，查看是否生成了对应钱包的“keystore”文件，如图 5-5 所示，从中可以看到，已经成功生成了。双击打开生成的文件，还能看到以太坊钱包生成的地址信息。

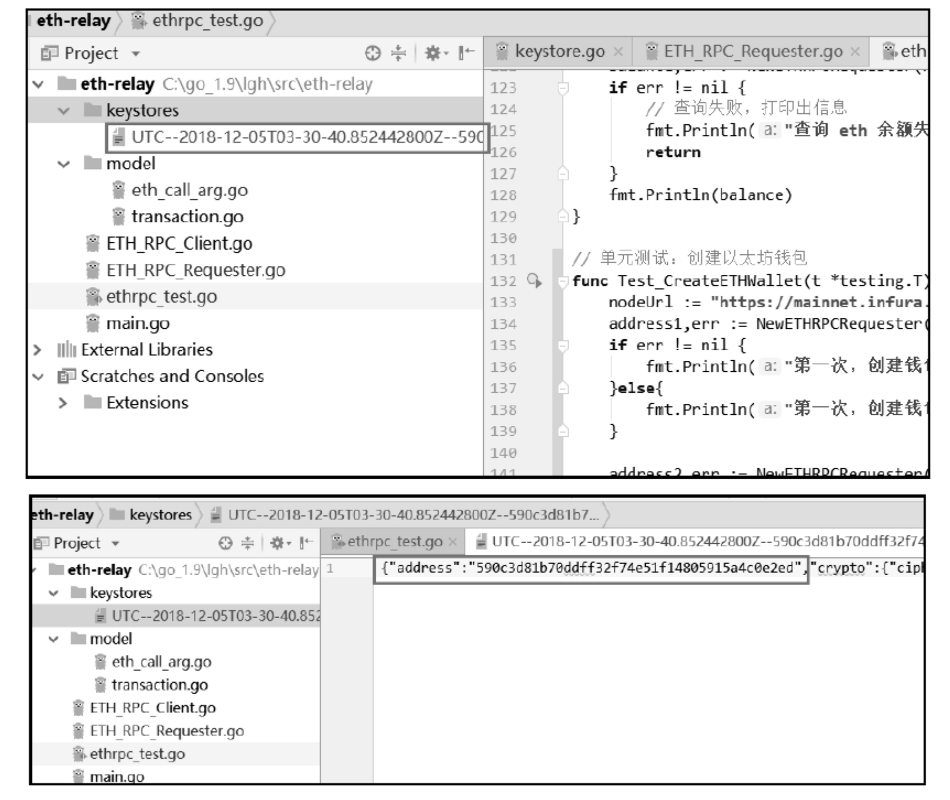



图 5-5　查看所生成钱包对应的 keystore 文件

至此，我们成功地生成了一个以太坊钱包，如果需要将这个钱包导入其他的钱包软件中使用，只需在钱包软件中选择导入“keystore”，再将上述“keystore”文件中的内容复制并粘贴进去，最后输入创建钱包时所设置的密码即可。

**提　示**

用来保存所有被创建钱包的“keystore”文件的文件夹“keystores”，实际上相当于存储了用户的钱包信息。如果将这里的某个 keystore 文件加上用户在创建它时所设置的密码，就能恢复出这个 keystore 文件所对应的私钥。这就是现在的中心化交易所存储用户钱包数据的一般做法。

# 5.2　实现以太坊交易

要实现“以太坊交易”功能，一般在前端通过编写代码就能达到目的。例如，在移动 App 或者网站上依靠 JavaScript 语言就能实现以太坊的交易功能，而不需要依赖后端服务来实现。但是，在中心化交易所里，一些交易功能往往需要在服务器端实现。例如，“用户提现”功能就是从公链上将资产转账到用户的钱包地址里。当然，通过客服向钱包地址转账也是可以的，但是当用户发起“提现”请求的量很大时，人为操作不仅效率低下，还需要专门招聘交易管理人员。除了交易所的“用户提现”功能之外，在其他 DApp 中还存在很多需要在服务器端实现交易的情况。例如，基于“ERC721”协议标准发布的个体类代币，如以太猫等的转让（转账）功能就是在服务器端进行的，转让是通过触发交易来实现的。

本节我们主要介绍如何使用“sendRawTransaction”接口来实现以太坊交易功能。

## 5.2.1　以太坊交易的原理

在实现交易函数之前，我们先要了解以太坊整个交易的流程：客户端签名交易、发起交易数据到以太坊节点、服务器端对交易数据的校验以及最终交易被添加到订单池的交易队列。下面我们通过这个流程对交易的原理进行讲解。

### 1.发送数据

第一步，从参数组装到发送给以太坊节点，这个过程需要对参数进行私钥签名。这个私钥的提供者对应传参中的“from”地址用户，私钥签名函数在“go-ethereum”源码中的“keystore.go”文件已提供了，如图 5-6 所示。其中，“*types.Transaction”就是待签名的交易数据结构体。

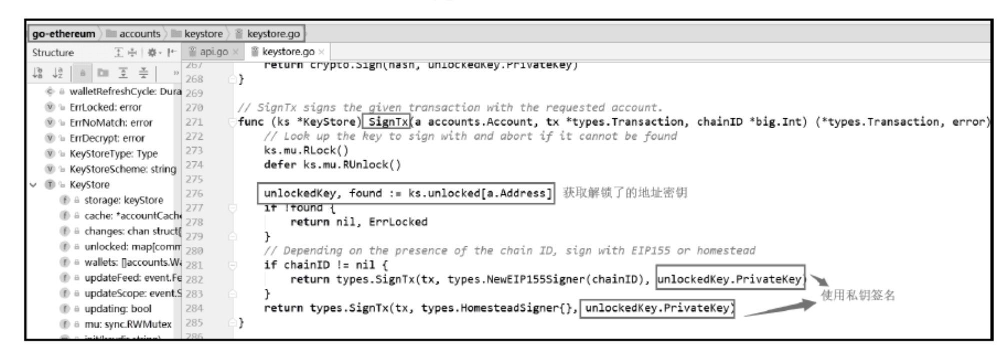



图 5-6　以太坊 Go 版本源码中签名交易的函数

签名成功后，将会对交易结构体（见图 5-7）中的“V”“S”“R”3 个字段进行赋值，生成的签名数据也是由它们存储的，在交易被提交到了节点后，节点的“验签”阶段，也是根据这 3 个字段进行的。

```go
type txdata struct {
	AccountNonce uint64          `json:"nonce"    gencodec:"required"`
	Price        *big.Int        `json:"gasPrice" gencodec:"required"`
	GasLimit     uint64          `json:"gas"      gencodec:"required"`
	Recipient    *common.Address `json:"to"       rlp:"nil"` // nil means contract creation
	Amount       *big.Int        `json:"value"    gencodec:"required"`
	Payload      []byte          `json:"input"    gencodec:"required"`

	// Signature values
	V *big.Int `json:"v" gencodec:"required"`
	R *big.Int `json:"r" gencodec:"required"`
	S *big.Int `json:"s" gencodec:"required"`

	// This is only used when marshaling to JSON.
	Hash *common.Hash `json:"hash" rlp:"-"`
}
```

签名时候进行赋值的三个重要字段

图 5-7　以太坊 Go 版本源码中的 txdata 结构体

第二步，将签好名的结构体数据进行“RLP 序列化”操作。同样地，这个操作所要使用的函数在“go-ethereum”源码中也提供了，对应于“rlp”依赖包下“encode.go”文件的“EncodeToBytes”函数，如图 5-8 所示。其中的“val”参数是一个泛型，在当前的情况中，它是签好名的结构体。

```go
// EncodeToBytes returns the RLP encoding of val.
// Please see the documentation of Encode for the encoding rules.
func EncodeToBytes(val interface{}) ([]byte, error) {
	eb := encbufPool.Get().(*encbuf)
	defer encbufPool.Put(eb)
	eb.reset()
	if err := eb.encode(val); err != nil {
		return nil, err
	}
	return eb.toBytes(), nil
}
```

val 入参就是我们签名好的结构体

图 5-8　以太坊 Go 版本源码中的 RLP 序列化函数

“RLP（递归长度前缀）”是一种编码方式，提供了一种适用于任意二进制数据数组的编码，已经成为以太坊中对对象进行序列化的主要编码方式。RLP 的唯一目标是解决结构体的编码问题，对基本数据类型（比如字符串、整数型、浮点型）的编码则交给更高层的协议处理，以太坊中要求数字必须是一个大端字节序的（Big-Endian）、没有零占位的存储格式。例如，“汉”这个字的 Unicode 编码是 6C49，在存储的时候，如果将 6C 放在前面，49 放在后面，就是大端字节序，因为 6C 的十进制形式比 49 的十进制要大。没有零占位就是不出现 0 位。例如，6C049 有一个零，就不满足没有零占位的要求。

关于“RLP”编码，这里有一篇很好的文章可以参考：

https://segmentfault.com/a/1190000011763339

## 2. 解读数据

上述签名并进行“RLP”编码之后的数据，将会被发送到以太坊的节点程序中。在节点程序中，它会对每一笔提交过来的交易进行数据的一系列校验。整个校验分为有 3 个步骤：RLP 反序列化、参数校验和订单池相关的判断。

（1）RLP 反序列化

第一步，将接收到的校验数据，进行“RLP 反序列化”操作，可以使用“go-ethereum”源码中“rlp”依赖包下“decode.go”文件中的“DecodeBytes”函数来进行。如图 5-9 所示。

```
go-ethereum > internal > ethapi > api.go

1232 }
1233
1234 // SendRawTransaction will add the signed transaction to the transaction pool.
1235 // The sender is responsible for signing the transaction and using the correct n
1236 func (s *PublicTransactionPoolAPI) SendRawTransaction(ctx context.Context, encode
1237 	tx := new(types.Transaction)
1238 	if err := rlp.DecodeBytes(encodedTx, tx); err != nil {
1239 		return common.Hash{}, err
1240 	}
1241 	return submitTransaction(ctx, s.b, tx)
1242 }
1243
```

RLP 反序列化

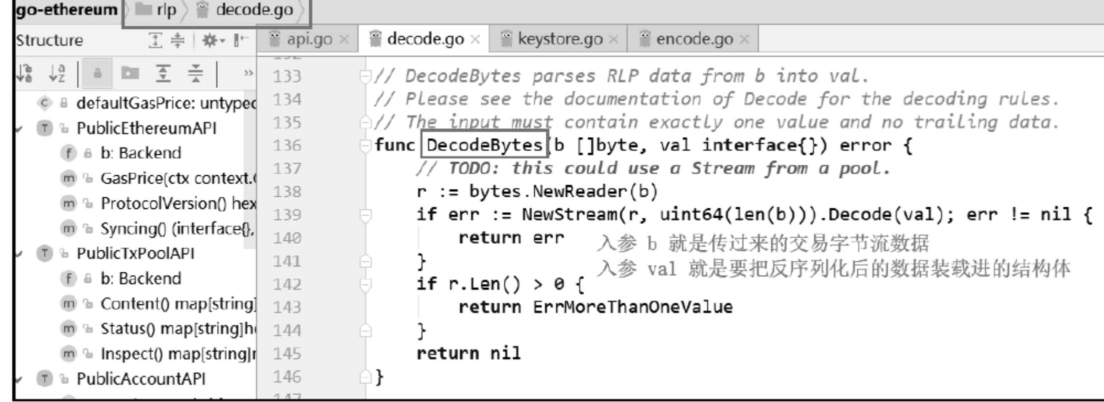



图 5-9　以太坊 Go 版本源码中的交易接收接口函数和 RLP 反序列化函数

（2）参数校验

第二步，对“RLP 反序列化”后恢复的交易数据进行校验。校验又分两部分：第一部分是基础数据的校验，例如燃料费“Gas”不能太低等；第二部分是对“V”“S”“R”的签名校验。

- 数据的基础校验，主要是一些范围限制及格式限制校验，包含数据量不能太大、防止“DDos”攻击、“Gas”值不能超过当前节点所设的最大值和“Nonce”值不能比已经成功过交易的序列值还低等，如图 5-10 所示。初步的校验工作在函数“validateTx”中进行。

```go
// validateTx checks whether a transaction is valid according to the conser
// rules and adheres to some heuristic limits of the local node (price and
func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {
	// Heuristic limit, reject transactions over 32KB to prevent DOS attack
	// 防止 ddos 攻击
	if tx.Size() > 32*1024 {      检查数据量
		return ErrOversizedData
	}
	// Transactions can't be negative. This may never happen using RLP deco
	// transactions but may occur if you create a transaction using the RPC
	if tx.Value().Sign() < 0 {
		return ErrNegativeValue
	}
	// Ensure the transaction doesn't exceed the current block limit gas.
	if pool.currentMaxGas < tx.Gas() {      检查 gas 是否超过最大值
		return ErrGasLimit
	}
	// Make sure the transaction is signed properly
	from, err := types.Sender(pool.signer, tx)
	if err != nil {
		return ErrInvalidSender
	}
	// Drop non-local transactions under our own minimal accepted gas price
	local = local || pool.locals.contains(from) // account may be local eve
	if !local && pool.gasPrice.Cmp(tx.GasPrice()) > 0 {
		return ErrUnderpriced
	}
	// Ensure the transaction adheres to nonce ordering
	if pool.currentState.GetNonce(from) > tx.Nonce() {
```

图 5-10　数据的基础校验

在基础数据校验阶段，会抛出以太坊交易的常见错误信息，如果发现有错误信息返回，就可以到对应的检测代码行进行排查。常见的以太坊交易错误信息如图 5-11 所示。

```go
var (
	// ErrInvalidSender is returned if the transaction contains an invalid signature.
	ErrInvalidSender = errors.New( text: "invalid sender")

	// ErrNonceTooLow is returned if the nonce of a transaction is lower than the
	// one present in the local chain.
	ErrNonceTooLow = errors.New( text: "nonce too low")

	// ErrUnderpriced is returned if a transaction's gas price is below the minimum
	// configured for the transaction pool.
	ErrUnderpriced = errors.New( text: "transaction underpriced")

	// ErrReplaceUnderpriced is returned if a transaction is attempted to be replaced
	// with a different one without the required price bump.
	ErrReplaceUnderpriced = errors.New( text: "replacement transaction underpriced")

	// ErrInsufficientFunds is returned if the total cost of executing a transaction
	// is higher than the balance of the user's account.
	ErrInsufficientFunds = errors.New( text: "insufficient funds for gas * price + value")

	// ErrIntrinsicGas is returned if the transaction is specified to use less gas
	// than required to start the invocation.
	ErrIntrinsicGas = errors.New( text: "intrinsic gas too low")

	// ErrGasLimit is returned if a transaction's requested gas limit exceeds the
	// maximum allowance of the current block.
	ErrGasLimit = errors.New( text: "exceeds block gas limit")
```

图 5-11　以太坊交易校验阶段的常见错误

- 签名信息的校验。这里所使用的是椭圆曲线算法“secp256k1”提供的方法，椭圆曲线算法“secp256k1”支持根据私钥导出公钥。对应的校验流程如图 5-12 所示，在“validateTx”函数内的“types.Sender”开始签名的校验。

```
go-ethereum 〉 core 〉 tx_pool.go
api.go × | decode.go × | backend.go × | api_backend.go × | tx_pool.go × | transaction_signing.go × | signatu

557  // validateTx checks whether a transaction is valid according to the consensus
558  // rules and adheres to some heuristic limits of the local node (price and size).
559  func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {
560      // Heuristic limit, reject transactions over 32KB to prevent DOS attacks
561      // 防止 ddos 攻击
562      if tx.Size() > 32*1024 {
563          return ErrOversizedData
564      }
565      // Transactions can't be negative. This may never happen using RLP decoded
566      // transactions but may occur if you create a transaction using the RPC.
567      if tx.Value().Sign() < 0 {
568          return ErrNegativeValue
569      }
570      // Ensure the transaction doesn't exceed the current block limit gas.
571      if pool.currentMaxGas < tx.Gas() {
572          return ErrGasLimit
573      }
574      // Make sure the transaction is signed properly
575      from, err := types.Sender(pool.signer, tx)          进入签名的校验
576      if err != nil {
577          return ErrInvalidSender
578      }
579      // Drop non-local transactions under our own minimal accepted gas price
```

图 5-12　validateTx 函数内调用 types.Sender 函数进行交易的校验

“types.Sender”函数的机制是先从缓存中检测，判断缓存中是否已经存在相同的签名信息，如图 5-13 所示。如果存在记录，就会直接返回校验结果。如果不存在，就会走完整的校验流程。缓存机制避免了重复操作，能在一定程度上提高代码的执行效率。

```
go-ethereum 〉 core 〉 types 〉 transaction_signing.go
api.go × | decode.go × | backend.go × | api_backend.go × | tx_pool.go × | transaction_signing.go × | signat

64
65  // Sender returns the address derived from the signature (V, R, S) using secp256k1
66  // elliptic curve and an error if it failed deriving or upon an incorrect
67  // signature.
68  //
69  // Sender may cache the address, allowing it to be used regardless of
70  // signing method. The cache is invalidated if the cached signer does
71  // not match the signer used in the current call.
72  func Sender(signer Signer, tx *Transaction) (common.Address, error) {
73      if sc := tx.from.Load(); sc != nil {
74          sigCache := sc.(sigCache)
75          // If the signer used to derive from in a previous
76          // call is not the same as used current, invalidate
77          // the cache.
78          if sigCache.signer.Equal(signer) {
79              return sigCache.from, nil
80          }
81      }
82
83      addr, err := signer.Sender(tx)          进入此处
84      if err != nil {
85          return common.Address{}, err
86      }
87      tx.from.Store(sigCache{signer: signer, from: addr})
88      return addr, nil
89  }
```

图 5-13　types.Sender 函数内部的实现代码

在“recoverPlain”函数中将会先由“S”“V”“R”3 个参数组合成签名信息，然后由“crypto.Ecrecover”恢复出私钥对应的公钥，最后由公钥得出地址，如图 5-14 所示。

```
go-ethereum > core > types > transaction_signing.go
api.go | decode.go | backend.go | api_backend.go | tx_pool.go | transaction_signing.go

125     var big8 = big.NewInt( x: 8)
126
127     func (s EIP155Signer) Sender(tx *Transaction) (common.Address, error) {
128         if !tx.Protected() {
129             return HomesteadSigner{}.Sender(tx)
130         }
131         if tx.ChainId().Cmp(s.chainId) != 0 {
132             // 默认启动节点 s.chainId = 1，也就是主网节点 chanId
133             return common.Address{}, ErrInvalidChainId
134         }
135         V := new(big.Int).Sub(tx.data.V, s.chainIdMul)
136         V.Sub(V, big8)                                   传入S V R 三参数
137         // 进入 recoverPlain
138         return recoverPlain(s.Hash(tx), tx.data.R, tx.data.S, V, homestead: true)
139     }
```

```
go-ethereum > core > types > transaction_signing.go
api.go | tx_pool.go | transaction_signing.go | signature_cgo.go | secp256.go | crypto.go | encode.go

222     }
223
224     func recoverPlain(sighash common.Hash, R, S, Vb *big.Int, homestead bool) (common.Address
225         if Vb.BitLen() > 8 {
226             return common.Address{}, ErrInvalidSig
227         }
228         V := byte(Vb.Uint64() - 27)
229         if !crypto.ValidateSignatureValues(V, R, S, homestead) { // 检查 V R S 的格式是否正确
230             return common.Address{}, ErrInvalidSig
231         }
232         // encode the snature in uncompressed format
233         r, s := R.Bytes(), S.Bytes()
234         sig := make([]byte, 65)
235         copy(sig[32-len(r):32], r)
236         copy(sig[64-len(s):64], s)
237         sig[64] = V
238         // recover the public key from the snature
239         pub, err := crypto.Ecrecover(sighash[:], sig)     推出公钥
240         if err != nil {
241             return common.Address{}, err
242         }
243         if len(pub) == 0 || pub[0] != 4 {
244             return common.Address{}, errors.New( text: "invalid public key")
245         }
246         var addr common.Address
247         copy(addr[:], crypto.Keccak256(pub[1:])[12:])     由公钥 pub 得出地址
248         return addr, nil
249     }
```

图 5-14　验签过程的函数调用

“crypto.Ecrecover”函数中的“secp256k1.RecoverPubkey”就是椭圆曲线算法恢复公钥的实例函数。该函数调用了基于“C 语言”的椭圆曲线算法实现库中的“secp256k1_ext_ecdsa_recover”函数，最终达到恢复的目的，如图 5-15 所示。

```
go-ethereum > crypto > signature_cgo.go
api.go | tx_pool.go | transaction_signing.go | signature_cgo.go | secp256.go | crypto.go

29
30      // Ecrecover returns the uncompressed public key that created the given signa
31      func Ecrecover(hash, sig []byte) ([]byte, error) {
32          return secp256k1.RecoverPubkey(hash, sig)   recover 恢复，pubkey 公钥
33      }
34
```

图 5-15　验签操作最终所调用的函数

（3）订单池相关的判断

最后一步是订单池相关的判断。这个校验过程分为两个小步骤，其中涉及的订单队列（Queue）和等待列表（Pending）都是订单池中的组成者成员。在以太坊“Go”版本的源码中，它们对应的

数据结构是“Map”。

① 判读订单池订单队列（Queue）是否已满，因为订单池的队列是可以被设置长度的，在订单队列满了的情况下，以太坊会将新进来的交易订单和队列中燃料费最低的一笔交易进行比较，然后将燃料费最低的一笔交易从订单池移出并抛弃。

由这一点我们可以知道，以太坊的订单池订单队列中的交易是根据燃料费高低，按照从高到低的顺序排队的，处于队尾的订单容易被抛弃。当以太坊拥堵的时候，如果新交易所设置的燃料费不够高，可能连排队的队列都不能进入。

图 5-16 所示是上述过程的大致流程图。

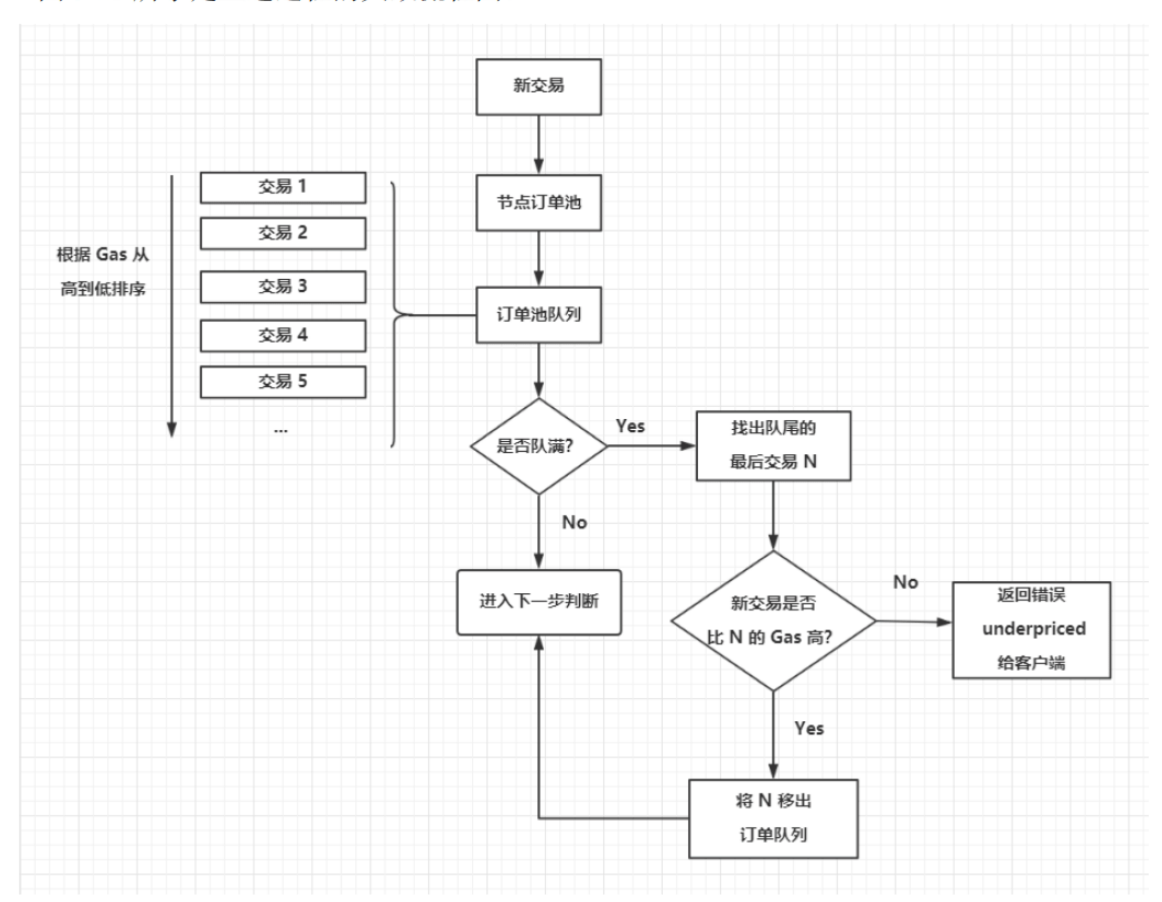



图 5-16　交易在订单池队列要经历的流程

对应的判断代码以及每个细节的注释已经在图 5-17 中给出。

② 新交易订单是否已经在“pending”（等待或挂起）列表中。以太坊的“pending”列表是用来存储那些已经从交易队列中被取出的交易项，这些交易是区块打包的候选交易。因为位于等待列表的交易处于等待状态，它们有可能会被重新提交过来，对于被重新提交过来的交易，以太坊的做法是将刚提交的和已经存在的交易的燃料费进行比较，如果新交易的燃料费比旧交易的燃料费高，就进行替换，如果不高，就返回错误给客户端。默认的燃料费替换规则是：新交易的燃料费要比旧交易的燃料费大于或等于 110%才可替换。

```
go-ethereum 〉 core 〉 tx_pool.go 〉
api.go × | tx_pool.go × | tx_list.go × | transaction_signing.go × | signature_cgo.go × | secp256.go × | crypto.go × | encode.go ×

623         log.Trace( msg: "Discarding invalid transaction",  ctx: "hash", hash, "err", err)
624         invalidTxCounter.Inc(1)
625         return false, err
626     }
627     // If the transaction pool is full, discard underpriced transactions
628     // 下面判断池中 tx 总数是不是满了，这里的 tx 是队列中，不是 pending 状态的
629     // pool.priced 是一个基于当前 pool 全局的、根据价格排序存储 tx 的对象，内部有一个 tx 数组
630     // pool.price.all，是和 pool.all 的地址一样的，传的是指针
631     if uint64(pool.all.Count()) >= pool.config.GlobalSlots+pool.config.GlobalQueue {
632         // If the new transaction is underpriced, don't accept it
633         if !local && pool.priced.Underpriced(tx, pool.locals) {     gas 价格判断
634             // 在还没被矿工处理的 pool.all.txs 中，已经满了的情况，还继续有 tx 过来，就进入和 under price
635             // 进入这里就是比 pool.all 中最低价的还要小，所以下面直接抛出价格过低不处理的错误，pool.all 与 p
636             log.Trace( msg: "Discarding underpriced transaction",  ctx: "hash", hash, "price", tx.GasPri
637             underpricedTxCounter.Inc(1)
638             return false, ErrUnderpriced     新交易的 gas 不够高，返回错误给客户端
639         }
640         // New transaction is better than our worse ones, make room for it
641         // 首次进入：
642         //      (all.Count == GlobalSlots + GlobalQueue），刚好多出一个
643         //      A = int(pool.config.GlobalSlots+pool.config.GlobalQueue-1)
644         //      pool.all.Count()- A = 1  刚好需要移除一个
645         drop := pool.priced.Discard(
646             pool.all.Count()-int(pool.config.GlobalSlots+pool.config.GlobalQueue-1),
647             pool.locals)                         移除队列最后的交易
648
649         for _, tx := range drop {
650             log.Trace( msg: "Discarding freshly underpriced transaction",  ctx: "hash", tx.Hash(), "pric
651             underpricedTxCounter.Inc(1)
652             pool.removeTx(tx.Hash(),  outofbound: false)
653         }
654         // 上面的 drop 移除了多出的后，下面就能正常添加
655     }
```

图 5-17　订单池队列中对新交易入队前的判断源码

图 5-18 是上述第②点对应的大致流程图。

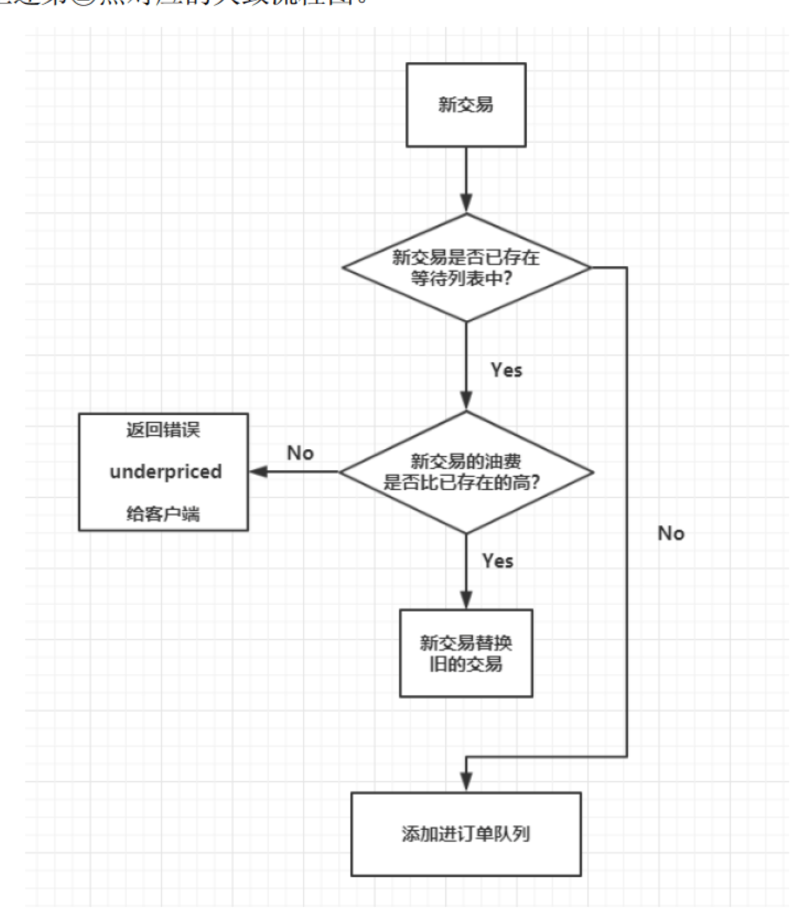



图 5-18　新交易进入到 pending 等待列表时的流程

对应的判断代码如下，每个细节的注释已经在图中给出。

判断一笔交易是否已经处于当前节点的“pending”等待列表的两个条件是：

- 当前交易发送者“from”已经在节点程序中并有其对应的“pending”等待列表。
- 当前交易发送者“from”的等待列表中有和当前新交易相同“nonce”序列号的交易。

图 5-19 所示是 pending 等待列表添加交易的流程源码。

```
go-ethereum 〉 core 〉 tx_pool.go
api.go × | tx_pool.go × | events.go × | tx_list.go × | transaction_signing.go × | signature_cgo.go × | secp256
promoteTx                                        Match Case  Words  Regex
656     // If the transaction is replacing an already pending one, do directly
657     from, _ := types.Sender(pool.signer, tx) // already validated
658     if list := pool.pending[from]; list != nil && list.Overlaps(tx) {
659
660                 根据发送者地址 from 来查询出该地址的交易等待列表
661         // Nonce already pending, check if required price bump is met
662         // pending 重发
663         inserted, old := list.Add(tx, pool.config.PriceBump) // PriceBump 默认是
664         if !inserted {
665             pendingDiscardCounter.Inc(1)      找出 nonce 相同的上一条交易
666             return false, ErrReplaceUnderpriced
667         }
668         // New transaction is better, replace old one
669         if old != nil {
670             pool.all.Remove(old.Hash())
671             pool.priced.Removed()
672             pendingReplaceCounter.Inc(1)  替换 gas 低的交易
673         }
674         pool.all.Add(tx)
675         pool.priced.Put(tx)
676         pool.journalTx(from, tx)
677
678         log.Trace( msg: "Pooled new executable transaction",  ctx: "hash", hash, "
679
680         // We've directly injected a replacement transaction, notify subsystems
681         go pool.txFeed.Send(NewTxsEvent{ Txs: types.Transactions{tx}})
682
683         return old != nil, nil
684     }                                          添加到交易的队列
685     // New transaction isn't replacing a pending one, push into queue
686     replace, err := pool.enqueueTx(hash, tx)
687     if err != nil {
```

```
go-ethereum 〉 core 〉 tx_list.go
api.go × | tx_pool.go × | events.go × | tx_list.go × | transaction_signing.go × | signature_cgo.go × | secp256.go ×
249 // If the new transaction is accepted into the list, the lists' cost and gas
250 // thresholds are also potentially updated.
251 func (l *txList) Add(tx *types.Transaction, priceBump uint64) (bool, *types.Transaction
252     // If there's an older better transaction, abort
253     old := l.txs.Get(tx.Nonce())  根据 nonce 去取出
254     if old != nil {
255         threshold := new(big.Int).Div(new(big.Int).Mul(old.GasPrice(), big.NewInt(100+i
256         // Have to ensure that the new gas price is higher than the old gas
257         // price as well as checking the percentage threshold to ensure that
258         // this is accurate for low (Wei-level) gas price replacements
259         if old.GasPrice().Cmp(tx.GasPrice()) >= 0 || threshold.Cmp(tx.GasPrice()) > 0 {
260             return false, nil
261         }
262     }
263     // Otherwise overwrite the old transaction with the current one
264     l.txs.Put(tx) // 添加到队列里面
265     if cost := tx.Cost(); l.costcap.Cmp(cost) < 0 {
266         l.costcap = cost
267     }
268     if gas := tx.Gas(); l.gascap < gas {
269         l.gascap = gas
270     }
271     return true, old
272 }
```

图 5-19　pending 等待列表添加交易的流程源码

（4）解答

在结束交易订单池的两个判断后，如果当前交易是全新的，就将顺利地通过代码行“pool.enqueueTx(hash, tx)”，被添加到节点程序的订单池交易队列中，等待被添加到“pending”

列表中，再被从“pending”列表取出然后广播出去。

在这个过程中，请注意以下两个要点：

- 处于“pending”等待状态的交易，如果有相同的“nonce”序列号，就会引发节点程序对它们做进一步的判断，然后选择出燃料费最高的，替换掉燃料费低的。
- 节点“挂”掉了（断网了或者宕机了），还没被处理的交易就会丢失。当交易还在交易队列时且还没被广播出去，这时节点程序“挂”了，存放在内存的数据就丢失了。

还需要注意的是，对于“sendTransaction”接口而言，因为它本身是在节点程序中直接执行，而非被远程调用执行，相当于用户在控制台直接执行控制台交易命令，这类直接在控制台中发起的交易被称为本地交易，即“local Tx”。

相比于远程交易，本地的交易具有更高的权限，这种权限体现在下面几点：

- 不轻易被替换。
- 在尚未被移除的时候，会被持久化地存储于本地一个文件中。
- 在节点启动进行交易数据恢复的时候，优先从本地加载到本地交易。

本地交易在“发送数据”阶段的数据签名，将直接由当前在节点程序中解锁了的账户提供私钥进行签名，不需要我们编写代码进行签名。同时，也不需要进行“RLP 序列化”。在“解读数据”阶段，也没有“RLP 反序列化”操作。

此外，对于订单池中处于“pending”列表的交易在被打包进区块中的时候，还没被从“pending”列表中移除，只有这个区块成为合法区块后，区块中打包了的交易才会被从交易池中移除掉，如果交易被写进了分叉块，交易池中的交易也不会减少，而是等待重新打包。

## 5.2.2 以太坊 ETH 的交易

本节我们来介绍调用以太坊 ETH 交易接口函数的操作流程。

### 1. 解锁钱包

在实现对交易数据进行签名之前，要先对当前交易中作为交易发起者的地址进行解锁操作。所谓解锁，就是获取到地址对应的私钥。

解锁钱包的操作流程是，将发起地址的“keystore”文件结合当初设置的密码解析出私钥，将私钥数据放在内存中，待需要对数据进行签名的时候使用。

以前面“创建钱包”一节中所创建的钱包“590c3d81b70ddff32f74e51f14805915a4c0e2ed”为例，其对应的“keystore”文件的“json”数据如下所示：

{"address":"590c3d81b70ddff32f74e51f14805915a4c0e2ed","crypto":{"cipher":"aes-128-ctr","ciphertext":"e5e3ca8af03367e4952e4aabb8d88763ef97e243e9ba4d54c3959c3f7389cefa","cipherparams":{"iv":"88957de75b94cf4d1d40f0b9153b9d74"},"kdf":"scrypt","kdfparams":{"dklen":32,"n":262144,"p":1,"r":8,"salt":"ace2c1f81d594b9e5f034dd355f968c13e1127fbf2c5f539a7ddecfa7ae2d139"},"mac":"d11c4daf4204c7280180029feedf51b8bfe267eb3c9bb4cdfa9e48d7d937b243"},"id":"a6e843d2-067e-4715-bc7a-85bf9d4b8b23","version":3}

密码是：

13456aa

以太坊“go-ethereum”源码中的“keystore.go”文件同样也提供了解锁钱包的函数源代码，该函数名称为 Unlock，记得前面的“创建钱包”一节中所用到的主要函数也是来自于“keystore.go”文件，可以说“keystore.go”源码文件包含了钱包各项操作的源码，如图 5-20 所示。

```
path > src > github.com > ethereum > go-ethereum > accounts > keystore > keystore.go
keystore.go ×
312     return types.SignTx(tx, types.HomesteadSigner{}, key.PrivateKey)
313 }
314
315 // Unlock unlocks the given account indefinitely.
316 func (ks *KeyStore) Unlock(a accounts.Account, passphrase string) error {
317     return ks.TimedUnlock(a, passphrase,  timeout: 0)
318 }
319                         解锁钱包
```

图 5-20　解锁钱包函数的源代码

要使用“keystore.go”中的“Unlock”函数，首先要实例化一个“KeyStore”对象指针，它的实例化步骤和创建钱包是一样的，实例化的时候需要传入当初设置存储“keystore”文件的文件夹。执行解锁时，对应的代码便会进入到这个文件夹寻找对应地址的“keystore”文件，再结合密码完成解锁操作。

因为钱包解锁函数不属于提供给客户端接口的类别，所以在项目中创建一个名称为“tool”的文件夹，代表工具集合，在该文件夹内创建名为“wallet.go”的文件，如图 5-21 所示。

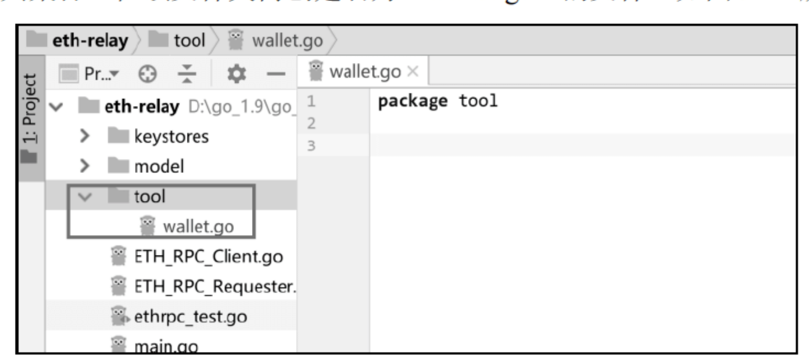



图 5-21　创建 wallet.go 文件

解锁钱包的完整代码如下，需要知道的一点是，所解锁出的私钥将会由“KeyStore”实例中的变量存储。因此在下面代码中需要一个全局变量的“KeyStore”实例变量“UnlockKs”来供程序使用。

```
// 全局地保存了已经解锁成功的钱包 map 集合变量
var ETHUnlockMap map[string]accounts.Account

// 全局地对应 keystore 实例
var UnlockKs *keystore.KeyStore

// 解锁以太坊钱包，传入钱包地址和对应的 keystore 密码
```

```
func UnlockETHWallet(keysDir string,address, password string) error {
    if UnlockKs == nil {
        UnlockKs = keystore.NewKeyStore(
            // 服务器端存储 keystore 文件的文件夹
            // 这些配置类的信息可以由配置文件指定
            keysDir,
            keystore.StandardScryptN,
            keystore.StandardScryptP)
        if UnlockKs == nil {
            return errors.New("ks is nil")
        }
    }
    unlock := accounts.Account{Address: common.HexToAddress(address)}
    // ks.Unlock 调用 keystore.go 的解锁函数，解锁出的私钥将存储在它里面的变量中
    if err := UnlockKs.Unlock(unlock, password); nil != err {
        return errors.New("unlock err : " + err.Error())
    }
    if ETHUnlockMap == nil {
        ETHUnlockMap = map[string]accounts.Account{}
    }
    ETHUnlockMap[address] = unlock // 解锁成功，存储
    return nil
}
```

解锁钱包的单元测试，将编写在与“tool.go”同级的测试文件中，新建“tool_test.go”文件，代表当前工具代码文件的测试文件。测试代码如下（参考图 5-22）：

```
func Test_UnlockETHWallet(t *testing.T) {
    address := "0x590c3d81b70ddff32f74e51f14805915a4c0e2ed"
    keysDir := "../keystores"
    // 第一次演示密码错误的情况
    err1 := UnlockETHWallet(keysDir,address, "789")
    if err1 != nil {
        fmt.Println("第一次解锁错误：", err1.Error())
    } else {
        fmt.Println("第一次解锁成功！")
    }
    // 第二次密码正确，解锁成功
    err2 := UnlockETHWallet(keysDir,address, "13456aa")
    if err2 != nil {
        fmt.Println("第二次解锁错误：", err1.Error())
    } else {
        fmt.Println("第二次解锁成功！")
    }
}
```

```go
package tool

import (
    "fmt"
    "testing"
)

func Test_UnlockETHWallet(t *testing.T) {

    // 第一次演示密码错误的情况
    err1 := UnlockETHWallet( address: "0x590c3d81b70ddff32f74e51f14805915a4c0e2ed", password: "789")
    if err1 != nil {
        fmt.Println( a: "第一次解锁错误：",err1.Error())
    }else{
        fmt.Println( a: "第一次解锁成功!")
    }
    // 第二次密码正确，解锁成功
    err2 := UnlockETHWallet( address: "0x590c3d81b70ddff32f74e51f14805915a4c0e2ed", password: "13456aa")
    if err2 != nil {
        fmt.Println( a: "第二次解锁错误：",err1.Error())
    }else{
        fmt.Println( a: "第二次解锁成功!")
    }
}
```

图 5-22　解锁钱包单元测试函数内的测试代码

运行结果如图 5-23 所示。

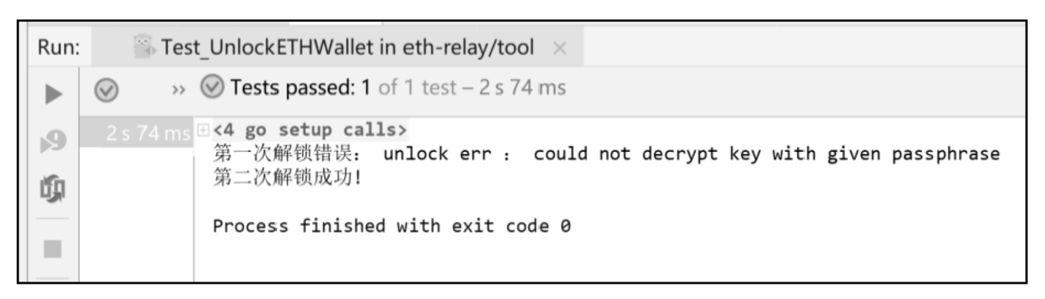



图 5-23　Test_UnlockETHWallet 解锁钱包单元测试函数的运行结果

## 2. 对数据进行签名

对数据进行签名所使用的 SignTx 函数也是由“keystore.go”源码文件提供的，如图 5-24 所示。

```go
    // Sign the hash using plain ECDSA operations
    return crypto.Sign(hash, unlockedKey.PrivateKey)
}

// SignTx signs the given transaction with the requested account.
func (ks *KeyStore) SignTx(a accounts.Account, tx *types.Transaction, chainID *big.Int) (*types.Transac
    // Look up the key to sign with and abort if it cannot be found
    ks.mu.RLock()                                    入参 tx 就是交易信息结构体
    defer ks.mu.RUnlock()

    unlockedKey, found := ks.unlocked[a.Address]
    if !found {
        return nil, ErrLocked
    }
    // Depending on the presence of the chain ID, sign with EIP155 or homestead
    if chainID != nil {
        return types.SignTx(tx, types.NewEIP155Signer(chainID), unlockedKey.PrivateKey)
    }
    return types.SignTx(tx, types.HomesteadSigner{}, unlockedKey.PrivateKey)
}
```

图 5-24　SignTx 函数的“keystore.go”源码

“SignTx”函数的第一个参数是传入当前解锁了的钱包地址，其内部的实现步骤是，首先根据钱包的地址获取对应的私钥，再使用私钥对交易信息结构体签名。第三个参数默认传入空值即可，它代表的是节点的 ID。

在“unlockedKey, found := ks.unlocked[a.Address]”中，“ks”的“unlocked”所保存的就是已经解锁了的钱包的私钥。

签名函数如下，其中“types.Transaction”是源码定义好的交易结构体，里面的每个参数和变量在“交易参数的说明”一节中都做过详细的说明。

```
type txdata struct {
   AccountNonce uint64       `json:"nonce"    gencodec:"required"`   // 交易序列号
   Price        *big.Int     `json:"gasPrice" gencodec:"required"`   // gasPrice
   GasLimit     uint64       `json:"gas"      gencodec:"required"`   // gasLimit
   // to 交易的接收者地址，"nil means contract creation" 的意思是，空意味着创建智能合约
   Recipient    *common.Address `json:"to"      rlp:"nil"` // nil means contract
creation
   Amount       *big.Int     `json:"value"    gencodec:"required"`   // 要交易的
代币数值
   Payload      []byte       `json:"input"    gencodec:"required"`   // data 参数

   // Signature values
   // 下面的 v、r、s 签名时会赋值，其中保存的是签名后生成的数据
   V *big.Int `json:"v" gencodec:"required"`
   R *big.Int `json:"r" gencodec:"required"`
   S *big.Int `json:"s" gencodec:"required"`

   Hash *common.Hash `json:"hash" rlp:"-"`
}

// 对交易数据结构体 types.Transaction 进行签名
func SignETHTransaction(address string,transaction *types.Transaction)
(*types.Transaction, error) {
   if UnlockKs == nil {
      return nil,errors.New("you need to init keystore first!")
   }
   account := ETHUnlockMap[address]
   if !common.IsHexAddress(account.Address.String()) {
      // 判断当前的地址钱包是否解锁了
      return nil,errors.New("account need to unlock first!")
   }
   return UnlockKs.SignTx(account,transaction,nil) // 调用签名函数
}
```

在签名函数的单元测试中，因为其前置的条件是要先解锁钱包，所以我们将其放在解锁钱包函数执行完之后再执行，最终的结果使用“json”的格式输出。我们观察输出后的“V”“R”“S”变量是否有值，有值则代表签名成功。代码如下：

```
func Test_UnlockETHWallet(t *testing.T) {
   address := "0x590c3d81b70ddf32f74e51f14805915a4c0e2ed"
   // 第一次演示密码错误的情况
   err1 := UnlockETHWallet(address,"789")
   if err1 != nil {
      fmt.Println("第一次解锁错误：",err1.Error())
   }else{
      fmt.Println("第一次解锁成功!")
   }
   // 第二次密码正确，解锁成功
   err2 := UnlockETHWallet(address,"13456aa")
   if err2 != nil {
```

```
    fmt.Println("第二次解锁错误：",err1.Error())
  }else{
    fmt.Println("第二次解锁成功!")
  }
  // 下面是签名的测试
  tx := types.NewTransaction(   //  创建一个测试用的交易数据结构体
    123,                        //  nonce 交易序列号
    common.Address{},           //  to 接收者地址
    new (big.Int).SetInt64(10), // value 数值
    1000,                       //  gasLimit
    new (big.Int).SetInt64(20), // gasPrice
    []byte("交易"))              //  data
  signTx,err := SignETHTransaction(address,tx)
  if err != nil {
    fmt.Println("签名失败!",err.Error())
    return
  }
  data,_ := json.Marshal(signTx)
  fmt.Println("签名成功\n",string(data))
}
```

运行结果如图 5-25 所示。

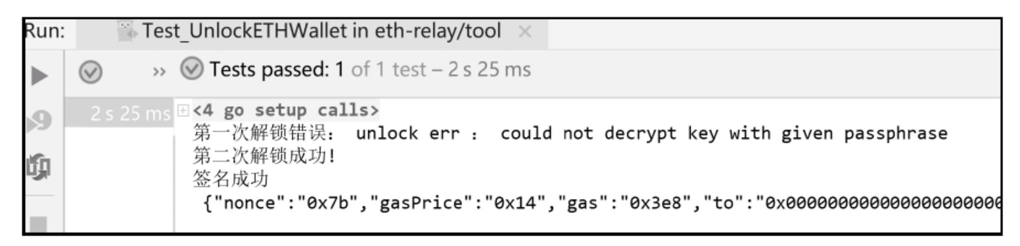



图 5-25　解锁钱包并对数据进行签名的运行结果

完整的“json”数据如下：


```
    {
        "nonce": "0x7b",
        "gasPrice": "0x14",
        "gas": "0x3e8",
        "to": "0x0000000000000000000000000000000000000000",
        "value": "0xa",
        "input": "0xe4baa4e69893",
        "v": "0x1c",
        "r":
"0x79583c15bc3464bb86e3e28aaf13e222025d7ef4744a270d44ce5d9d6e1629f",
        "s":
"0x5accb4bf5c8961b7e2adf922048d78da5fe3fd8e8444e62ba441fa793fa519dd",
        "hash":
"0xe449560dc61e79203fd5aaca4d197548e670477635834710276a22c5818af37f"
    }
```


### 3. 发送交易

在签名完成之后，调用以太坊的“eth_sendRawTransaction”接口发送交易数据，将交易发送到节点中去。

最后的一步是“RLP 序列化”，根据我们前面“发送数据”一节讲到的，我们知道该序列化数据的函数在源码“rlp”依赖包下的“encode.go”文件中，函数名称是“EncodeToBytes”。直接将签名好的交易数据进行传参调用，即可一步到位，如图 5-26 所示。

```
// 进行 rlp 序列化
txData, err := rlp.EncodeToBytes(tx)
if nil != err {
    return tx, err
}
```

图 5-26　调用 rlp 序列化函数

在“RLP 序列化”后，就可以使用“RPC”客户端请求者向以太坊节点进行“RPC”请求了，完整的交易函数如下：

```
// 发送交易，根据入参 transaction 的不同变量设置，达到发送不同种类的交易
func (r *ETHRPCRequester) SendTransaction(address string,transaction
*types.Transaction) (string,error) {
  // 对交易数据进行签名
  signTx,err := tool.SignETHTransaction(address,transaction)
  if err != nil {
    return "",fmt.Errorf("签名失败! %s",err.Error())
  }
  // rlp 序列化
  txRlpData, err := rlp.EncodeToBytes(signTx)
  if nil != err {
    return "", fmt.Errorf("rlp 序列化失败! %s",err.Error())
  }
  // 下面调用以太坊的 rpc 接口
  txHash := ""
  methodName := "eth_sendRawTransaction"
  err = r.client.client.Call(&txHash,methodName, common.ToHex(txRlpData))
  if err != nil {
    return "", fmt.Errorf("发送交易失败! %s",err.Error())
  }
  return txHash,nil // 返回交易 hash
}
```

实现代码存放在“ETH_RPC_Requester.go”文件中，之后就可以根据传入的交易结构体内部参数不同的设置而达到发起不同种类的交易。

下面我们根据上面完成的“SendTransaction”函数来分别封装实现以太坊 ETH 的转账交易函数以及非 ETH 类的代币转账函数。

### 4. nonce 管理器

发起交易的数据中需要传递“nonce”序列号参数，而且每笔新交易的“nonce”要求必须要比当前交易发起者最近一笔成功交易的“nonce”值要大。

下面再次列出“nonce”的作用，详细介绍见“Nonce 的作用”一节中的讲解。

（1）作为交易接口的参数。

（2）代表每次交易的序列号，方便节点程序处理被重复发起的交易。

（3）如果“nonce”比最近一笔成功交易的“nonce”要小，转账出错。

（4）如果“nonce”比最近一笔成功交易的“nonce”大了不止 1，那么这笔发起的交易就会长久处于队列之中，此时不是等待（pending）状态！在补齐了此“nonce”值到最近成功的那笔交易的“nonce”值之间的“nonce”值后，此笔交易就可以被执行。

（5）还处于队列中的交易，不考虑其他节点缓存广播的情况下，如果此时节点“挂”了，那么尚未被处理的交易将会丢失。

（6）处于 pending 等待状态的交易，如果具有相同的 nonce，就会引发节点程序对它们进一步的判断，然后选择燃料费最高的，替换掉燃料费低的。

因为“nonce”在交易中起到了上述十分重要的作用，而以太坊源码中并没有帮助我们管理“nonce”，所以需要我们自己实现一个“nonce”管理器，来计算当前交易发起者发起交易的时候应该将“nonce”的值设为多少才正确。这就是“nonce”管理器的主要作用。

实现“nonce”管理器主要使用以太坊接口中的“eth_getTransactionCount”接口，该接口的相关介绍见“重要接口的含义详解”一节。

查看以太坊的“RPC”接口文档可知，“eth_getTransactionCount”接口需要传入两个参数：第一个是当前要获取的“nonce”的以太坊地址值，在交易中，这个参数就是交易发起者的地址；第二个参数是区块号参数，该参数返回的结果是一个十六进制的“nonce”值，如图 5-27 所示。

**eth_getTransactionCount**

Returns the number of transactions *sent* from an address.

**Parameters**

1. `DATA`, 20 Bytes - address.
2. `QUANTITY|TAG` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

```
params: [
   '0x407d73d8a49eeb85d32cf465507dd71d507100c1',
   'latest' // state at the latest block
]
```

**Returns**

`QUANTITY` - integer of the number of transactions send from this address.

**Example**

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionCount","params":["0x407d73d8a49e

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

图 5-27　eth_getTransactionCount 接口的文档介绍

根据文档中提供的信息，我们到“ETH_RPC_Requester.go”文件中先实现“eth_getTransactionCount”接口的请求函数，代码如下：

```
// 获取地址的 nonce 值
func (r *ETHRPCRequester) GetNonce(address string) (uint64,error) {
   methodName := "eth_getTransactionCount"  // 指定接口名称
   nonce := ""
   // 因为我们要查询最新的，根据基于 etTransactionCount 情况下的区块号关系，选取 pending
   err := r.client.client.Call(&nonce,methodName, address,"pending")
   if err != nil {
      return 0, fmt.Errorf("发送交易失败! %s",err.Error())
   }
   n,_ := new(big.Int).SetString(nonce[2:],16) // 采用大数类型将十六进制的结果值转为十
进制的结果值
   return n.Uint64(),nil // 返回交易的哈希值
}
```

其中为什么第二个参数要选择“pending”，原因在“重要接口的含义详解”一节讲解“eth_getTransactionCount”接口时已有说明，下面再做一点补充：

eth_getTransactionCount 接口根据以太坊钱包地址获取基于当前钱包地址的交易序列号 Nonce。第二个传入的参数 genesis、pending 和 latest，分别对应下面的效果：

- 取 genesis 时，获取当前以太坊地址第一次发起交易时的 Nonce 序列号。
- 取 pending 时，获取当前以太坊地址提交的正处于 pending 状态等待被区块打包的交易订单所对应的 Nonce 序列号。请注意，如果当前所查询的地址没有处于 pending 的交易状态，那么它将返回与 latest 一样的 Nonce 号。
- 取 latest 时，获取当前以太坊地址当前提交了且被区块成功打包了的交易订单所对应的 Nonce 序列号加 1 的值。举个例子，地址 A 最后一笔成功交易对应的 Nonce 为 4，那么当调用接口传入该参数的时候，获取的结果是 5。

在 eth_getTransactionCount 中，Nonce 查询满足：pending ≥ latest ≥ genesis。

单元测试代码如下：

```
// 单元测试： 获取 nonce
func Test_GetNonce(t *testing.T) {
   nodeUrl := "https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b"
   address := "0x0D0707963952f2fBA59dD06f2b425ace40b492Fe"
   if address == "" || len(address) != 42 {
      // 这里演示在调用 rpc 接口函数时要先进行入参的合法性判断
      fmt.Println("非法的交易地址值")
      return
   }
   nonce,err := NewETHRPCRequester(nodeUrl).GetNonce(address)
   if err != nil {
      // 查询失败，打印出信息
      fmt.Println("查询 nonce 失败，信息是：",err.Error())
      return
   }
   fmt.Println(nonce)
}
```

运行结果如图 5-28 所示。

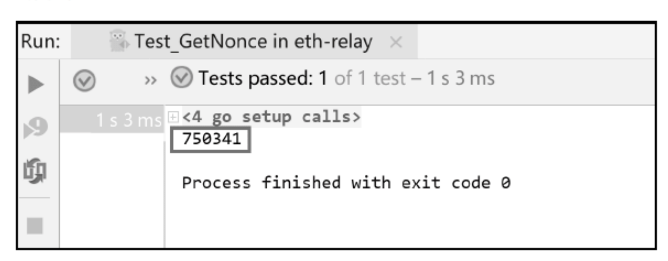



图 5-28　Test_GetNonce 单元测试函数的运行结果

在“GetNonce”函数的基础上，我们开始设计和实现“nonce”管理器。其原理是，假设“nonce”值不做“硬”存储则不会存放到数据库或者文件中，那么在程序首次运行发起交易时，会调用一次“GetNonce”函数，从以太坊节点网络中获取当前合理的“nonce”值。当发起了一笔交易，且成功获取了以太坊节点返回的交易哈希之后，我们就将“nonce”值进行加 1 的操作，并存放在内存中。接下来发起的其他交易中的“nonce”值会直接从内存中获取出来。当某次交易发送错误的时候，我们再次使用“GetNonce”函数获取一次节点中的“nonce”值，重发一次当前失败的交易，以此循环。大致流程图如图 5-29 所示。

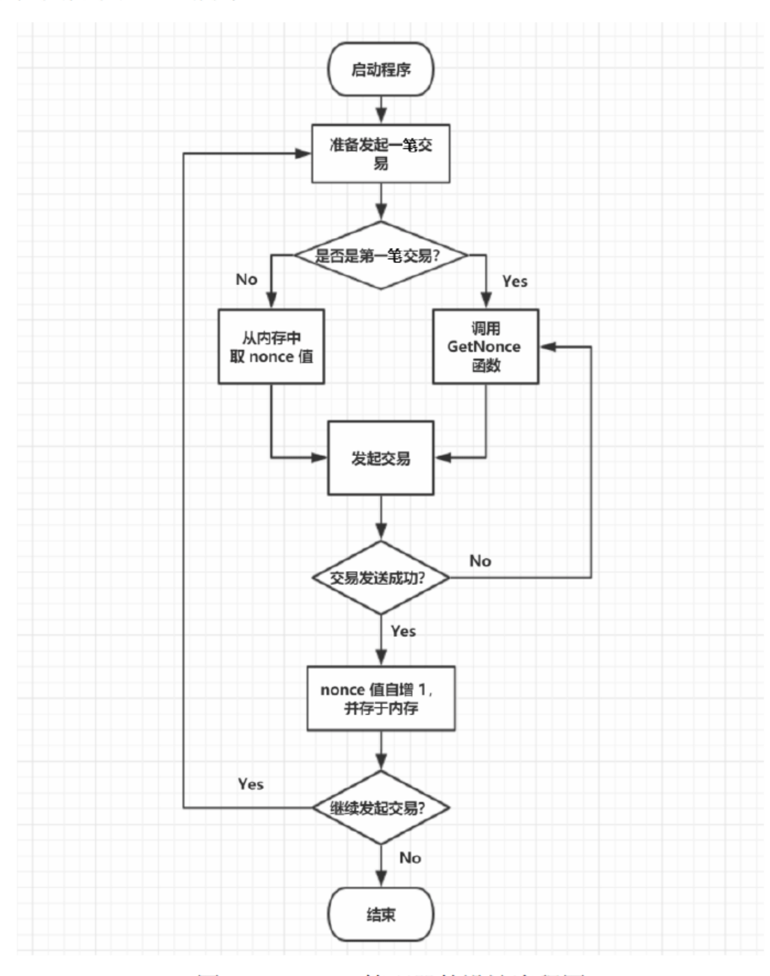



图 5-29　nonce 管理器的设计流程图

接下来在项目主文件夹下创建一个“nonce_manager.go”文件，用来存储“nonce”的管理实

现代码，如图 5-30 所示。

图 5-30 创建一个“nonce_manager.go”文件

代码如下：

```
// 管理器结构体
type NonceManager struct {
   // lock 是互斥锁，go 的 map 类型不是协程安全的，
   // 在读写 map 的时候，我们要考虑多协程并发的情况
   lock sync.Mutex

   // 采用整型大数来存储 nonce
   nonceMemCache map[string]*big.Int
}

func NewNonceManager() *NonceManager {
   return &NonceManager{
      lock: sync.Mutex{}, // 实例化互斥锁
   }
}
//  设置 nonce
func (n *NonceManager) SetNonce(address string,nonce *big.Int) {
   if n.nonceMemCache == nil {
      n.nonceMemCache = map[string]*big.Int{}
   }
   n.lock.Lock()          // 加锁
   defer n.lock.Unlock() // 当该函数执行完毕，进行解锁
   n.nonceMemCache[address] = nonce
}

// 根据以太坊地址获取 nonce
func (n *NonceManager) GetNonce(address string) *big.Int {
   if n.nonceMemCache == nil {
      n.nonceMemCache = map[string]*big.Int{}
   }
   n.lock.Lock()          // 加锁
   defer n.lock.Unlock() // 当该函数执行完毕，进行解锁
   return n.nonceMemCache[address]
}
```

```
// nonce 进行加 1 的操作
func (n *NonceManager) PlusNonce(address string) {
   if n.nonceMemCache == nil {
      n.nonceMemCache = map[string]*big.Int{}
   }
   n.lock.Lock()          // 加锁
   defer n.lock.Unlock() // 当该函数执行完毕，进行解锁
   oldNonce := n.nonceMemCache[address]
   newNonce := oldNonce.Add(oldNonce, big.NewInt(int64(1)))
   n.nonceMemCache[address] = newNonce
}
```

然后，对之前的“RPC”请求者进行部分修改，把“nonce”管理器的对象指针添加进去，作为请求者的一个变量，这样在使用请求者交易函数的时候才能使用对应的管理器，同时在请求者初始化的时候初始化“nonce”管理器。修改代码，如图 5-31 所示。

```
16  type ETHRPCRequester struct {
17      nonceManager *NonceManager // nonce 管理者实例
18      client       *ETHRPCClient // 小写字母开头，私有的 rpc 客户端
19  }
20
21  func NewETHRPCRequester(nodeUrl string) *ETHRPCRequester {
22      requester := &ETHRPCRequester{}
23      // 实例化 nonce 管理器
24      requester.nonceManager = NewNonceManager()
25      // 实例化 rpc 客户端
26      requester.client = NewETHRPCClient(nodeUrl)
27      return requester
28  }
```

图 5-31　修改“RPC”请求者的代码

最后在“发送交易”的函数后面，加上当交易的“hash”值正确返回时当前发起交易的地址“nonce”值加 1，这样才能确保在批量发起交易的业务场景中实现“nonce”值的正确增加。修改代码如下：

```
err = r.client.client.Call(&txHash, methodName, common.ToHex(txRlpData))
if err != nil {
   return "", fmt.Errorf("发送交易失败! %s", err.Error())
}
oldNonce := r.nonceManager.GetNonce(address)
if oldNonce == nil {
   r.nonceManager.SetNonce(address,new(big.Int).SetUint64(transaction.Nonce()))
}
r.nonceManager.PlusNonce(address)  // 成功后，当前用户内存的 nonce 值加 1
return txHash, nil // 返回交易的哈希值
```

下面将按照图 5-29 的流程图实现“nonce”管理器。

### 5. 发送 ETH 交易

综合前面各节的讲解，就可以实现专门用于进行以太坊 ETH 交易的转账函数了。

首先在“tool.go”文件中实现一个“value”与代币的“decimal”乘积函数，代码如下：

```
// 根据代币的 decimal 得出乘上 10^decimal 后的值
```

```
// value 是包含浮点数的，例如 0.5 个 ETH
func GetRealDecimalValue(value string,decimal int) string {
  if strings.Contains(value, ".") {
    // 小数
    arr := strings.Split(value, ".")
    if len(arr) != 2 {
      return ""
    }
    num := len(arr[1])
    left := decimal - num
    return arr[0] + arr[1] + strings.Repeat("0", left)
  } else {
    // 整数
    return value + strings.Repeat("0", decimal)
  }
}
```

之所以采用如上的函数实现，是因为代币的交易数值可以是浮点数。当为浮点数的时候，需要乘上“10^decimal”再转为大数形式，目前 Go 语言还没有现成的库函数可以使用，所以要自己编写函数。

最后是 ETH 交易函数的实现代码：

```
// 发送 ETH 交易，或称转账 ETH
// 参数分别是交易发起地址、交易接收地址、ETH 数量、燃料费设置
func (r *ETHRPCRequester) SendETHTransaction(fromStr, toStr, valueStr string,
gasLimit, gasPrice uint64) (string, error) {

  if !common.IsHexAddress(fromStr) || !common.IsHexAddress(toStr) {
    return "", errors.New("invalid address")
  }

  to := common.HexToAddress(toStr) // 将字符串类型的转为 address 类型的
  gasPrice_ := new(big.Int).SetUint64(gasPrice)

  // value 乘上 10^decimal，得出真实的转账值，ETH 单位精确到小数点后 18 位
  realV := tool.GetRealDecimalValue(valueStr, 18)
  if realV == "" {
    return "", errors.New("invalid value")
  }
  amount, _ := new(big.Int).SetString(realV, 10)

  // 获取 nonce
  nonce := r.nonceManager.GetNonce(fromStr)
  if nonce == nil {
    // nonce 不存在，开始访问节点来获取
    n, err := r.GetNonce(fromStr)
    if err != nil {
      return "", fmt.Errorf("获取 nonce 失败 %s", err.Error())
    }
    nonce = new(big.Int).SetUint64(n)
    r.nonceManager.SetNonce(fromStr,nonce) // 为当前的地址设置 nonce
  }
```

```
    // 构建 data，因为 eth 是交易转账类型，所以 data 是空的，我们设置空字符串即可
    data := []byte("")

    // 构建交易结构体
    transaction := types.NewTransaction(
        nonce.Uint64(),
        to,
        amount,
        gasLimit,
        gasPrice_,
        data)

    return r.SendTransaction(fromStr, transaction)
}
```

接下来进行 ETH 交易转账的单元测试，在测试中，我们使用之前在“获取链接”一节中的“Infura”所申请的测试节点 https://ropsten.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b，以及“获取测试币”一节中在水龙头网站所申请到的 ETH 测试代币进行测试。当然，如果读者具备可以使用的以太坊主网钱包且钱包里拥有 ETH 代币，当然也就可以使用主网的 ETH 代码进行测试。

此外，不要忘记了在发起交易之前，还要对发起者的钱包进行解锁，所以我们先得把代码中充当发起者的地址对应的“keystore”文件放入项目中的“keystores”文件夹中。如图 5-32 所示，新建一个“mykey.json”文件，并粘贴到交易发起者的“keystore”的“json”文件中。

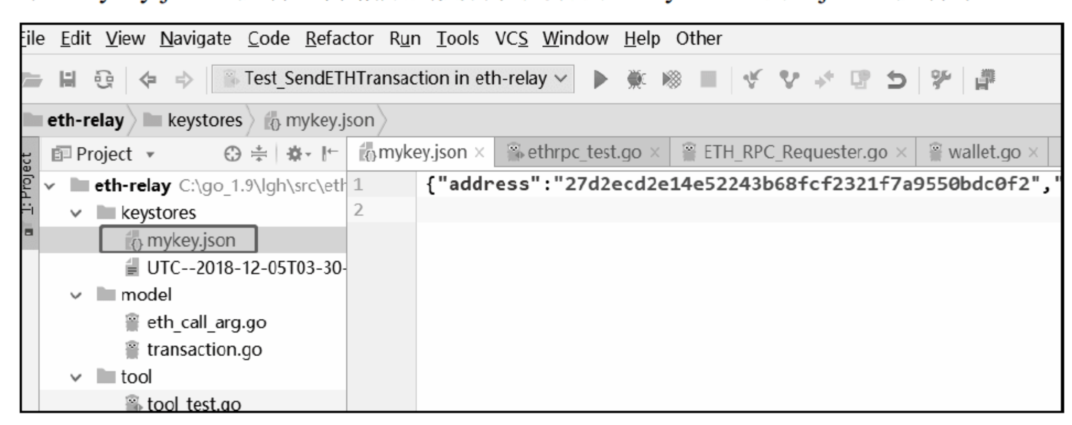



图 5-32　添加钱包的 keystore 文件到项目中

最终发送 ETH 交易的单元测试代码如下：

```
// 单元测试：转账 ETH
func Test_SendETHTransaction(t *testing.T) {
    nodeUrl := "https://ropsten.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b"
// ropsten 测试网络的节点链接
    from := "0x27d2ecd2e14e52243b68fcf2321f7a9550bdc0f2"  // 这个地址就是当初获取测试
代币的地址
    if from == "" || len(from) != 42 {
        // 这里演示在调用 rpc 接口函数时要先进行入参的合法性判断
        fmt.Println("非法的交易地址值")
        return
    }
    to := "0xd8CCEFDac5F30f06C62ed13383e9563C482630Bc"
    value := "0.2"  // 发送 0.2 个 ETH
```

```
    gasLimit := uint64(100000)
    gasPrice := uint64(36000000000)
    // 当前这笔交易消耗的燃料费最大值是 (gasLimit * gasPrice)/10^18 ETH
    err := tool.UnlockETHWallet("./keystores",from,"123aaaaa")  // 解锁钱包
    if err != nil {
        fmt.Println(err.Error())
        return
    }
    // 下面发起交易转账
    txHash, err :=
NewETHRPCRequester(nodeUrl).SendETHTransaction(from,to,value,gasLimit,gasPrice)
    if err != nil {
        // 转账失败，打印出信息
        fmt.Println("ETH 转账失败，信息是: ", err.Error())
        return
    }
    fmt.Println(txHash)  // 打印出当前交易的哈希值
}
```

运行结果如图 5-33 所示。




图 5-33　发送 ETH 交易后节点返回的 txHash

根据交易的哈希值，可以到以太坊区块链浏览器中查询验证这笔交易，查询链接及其结果如图 5-34 所示。

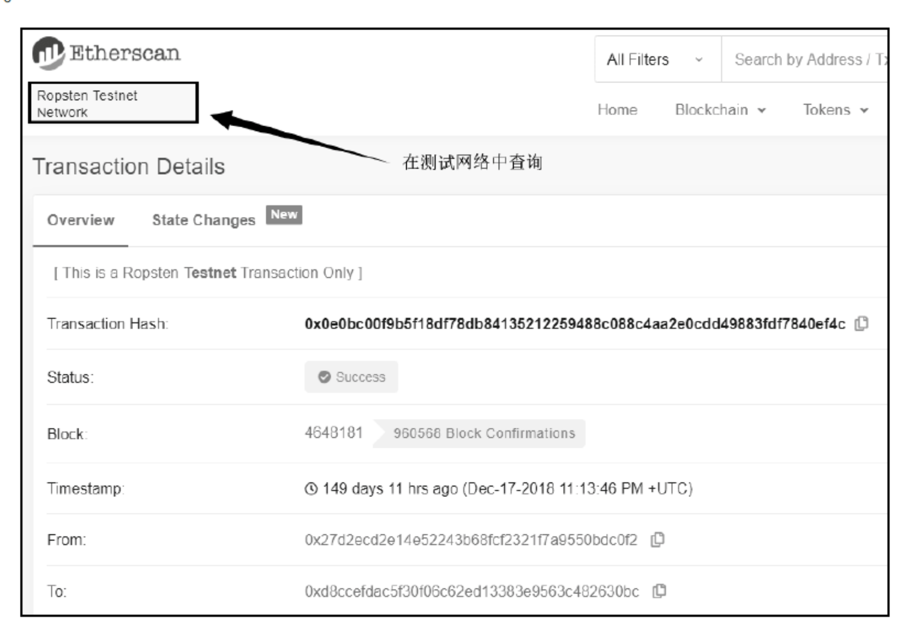



图 5-34　查询链接及其验证结果

可以看到，交易已经成功发送到以太坊的测试节点去了，对应的转账数值也是所设置的 0.2 个代币。

### 6. 发送 ERC20 代币交易

“ERC20”代币的转账函数和 ETH 的不同主要在于交易结构体参数中的“value”和“data”参数。

根据之前的“交易参数的说明”一节中的讲解，在“senRawTransaction”接口中要实现非 ETH 的交易，需要满足下面几个条件：

- “to”参数是对应的智能合约的地址。
- “value”参数为 0。
- “data”参数由“transfer”的“methodId”加上合约参数的特定的十六进制格式组成。

在上面的条件中，第一、二点容易实现，第三点需要编写一个转换函数，用来专门构建符合“ERC20”标准的“transfer”合约函数的“data”入参。转换函数的代码如下：

```
// 构建符合"ERC20"标准的"transfer"合约函数的"data"入参
func BuildERC20TransferData(value,receiver string,decimal int) string {

   realValue := GetRealDecimalValue(value,decimal)  // 将 value 乘上 10^decimal 的
格式
   valueBig, _ := new(big.Int).SetString(realValue, 10)

   // 按照 "交易参数的说明"小节中的讲解进行构建
   methodId := "0xa9059cbb" // "0xa9059cbb" 是 transfer 的 methodId
   param1  := common.HexToHash(receiver).String()[2:]  // 第一个参数，收款者地址
   param2  := common.BytesToHash(valueBig.Bytes()).String()[2:]
// 第二个参数，交易的数值
   return methodId + param1 + param2
}
```

要注意的是，上面的构建函数仅符合“ERC20”标准，对于非“ERC20”标准的每一份智能合约内的代币转账函数，要具体情况具体分析。例如，一个合约的转账代币的函数名称是“SendToken”，那么构建“data”的时候，要针对这份合约构建出符合该函数的“data”。

“ERC20”代币转账函数的代码如下：

```
// 发送 ERC20 代币交易，或称转账 ERC20 代币
// 参数分别是：
// 交易的发起地址，代币的合约地址，交易接收地址，代币数量，燃料费设置，代币的 decimal 值
func (r *ETHRPCRequester) SendERC20Transaction(
   fromStr, contact, receiver, valueStr string, gasLimit, gasPrice uint64, decimal
int) (string, error) {

   if  !common.IsHexAddress(fromStr) ||
      !common.IsHexAddress(contact) ||
      !common.IsHexAddress(receiver) {
      return "", errors.New("invalid address")
   }

   to := common.HexToAddress(contact) // 将合约 contact 字符串类型转为 address 类型
```

```
    gasPrice_ := new(big.Int).SetUint64(gasPrice)

    // 结构体中的 value 字段为 0
    amount := new(big.Int).SetInt64(0)

    // 获取 nonce
    nonce := r.nonceManager.GetNonce(fromStr)
    if nonce == nil {
        // nonce 不存在，开始访问节点获取
        n, err := r.GetNonce(fromStr)
        if err != nil {
            return "", fmt.Errorf("获取 nonce 失败 %s", err.Error())
        }
        nonce = new(big.Int).SetUint64(n)
        r.nonceManager.SetNonce(fromStr,nonce) // 为当前的地址设置 nonce
    }

    // 构建 data，真实的 value 转账数值由 data 携带
    data := tool.BuildERC20TransferData(valueStr, receiver, decimal)
    dataBytes := common.FromHex(data) // 使用以太坊提供的函数将十六进制数据转为字节

    // 构建交易结构体
    transaction := types.NewTransaction(
        nonce.Uint64(),
        to,
        amount,
        gasLimit,
        gasPrice_,
        dataBytes)

    return r.SendTransaction(fromStr, transaction)
}
```

“ERC20”代币转账的测试需要用到一份代币合约，根据第 4 章“智能合约的编写、发布、调用”一节中学到的知识，我们首先在以太坊测试网络中发布一份“ERC20”代币合约，要发布的合约是“实现 ERC20 代币智能合约”一节中的“MyToken.sol”示例，代币名称是“MFTC”。在 Mist 中先切换节点的网络到“Ropsten”测试网络模式，再到 Remix 提取合约的“Bytecode”，而后粘贴到 Mist 中进行发布。

当然也可以使用已经在测试网络“Ropsten”中发布了的测试“ERC20”智能合约，合约地址是 0x99BD856a01210D3B4b76A6f8c6fFf3eCdC485758。

单元测试代码如下：

```
// 单元测试：转账 ERC20 代币
func Test_SendERC20Transaction(t *testing.T) {
    // ropsten 测试网络的节点链接
    nodeUrl := "https://ropsten.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b"
    from    := "0x27d2ecd2e14e52243b68fcf2321f7a9550bdc0f2"
// 这个地址就是当初获取测试代币的地址
    if from == "" || len(from) != 42 {
        // 这里演示在调用 rpc 接口函数时先进行入参的合法性判断
        fmt.Println("非法的交易地址值")
        return
```

```
   }
   to  := "0x99BD856a01210D3B4b76A6f8c6fFf3eCdC485758"
// 在测试网络上发布的 MFTC 代币智能合约
   amount   := "10" // 转账 ERC20 代币的数值，10 个 MFTC
   decimal  := 18   // MFTC 代币单位精确到小数点后的位数
   receiver := "0xd8CCEFDac5F30f06C62ed13383e9563C482630Bc" // 接收者的以太坊地址
   gasLimit := uint64(50000)
   gasPrice := uint64(24000000000)
   // 当前这笔交易消耗的燃料费最大值是 (gasLimit * gasPrice) / 10^18 ETH
   err := tool.UnlockETHWallet("./keystores", from, "123aaaaa") // 解锁钱包
   if err != nil {
      fmt.Println(err.Error())
      return
   }
   // 下面发起转账交易
   txHash, err :=
      NewETHRPCRequester(nodeUrl).
      SendERC20Transaction(from, to,receiver,amount, gasLimit, gasPrice,decimal)
   if err != nil {
      // 转账失败，打印出信息
      fmt.Println("ETH 转账失败，信息是：", err.Error())
      return
   }
   fmt.Println(txHash) // 打印出当前交易的哈希值
}
```

运行单元测试转账后，可以看到控制台输出了交易的“hash”值（见图 5-35）：

0x2bfb41beab942f88cace64d2d4f8b85ad6b07468e22cb92db244443567adfaff




图 5-35　控制台输出了交易的“hash”值

根据这个“hash”值到以太坊测试网络“Ropsten”的区块链浏览器中查询，从页面中可以看到已经转账成功了，如图 5-36 所示。

至此，“ERC20”转账请求函数成功运行。

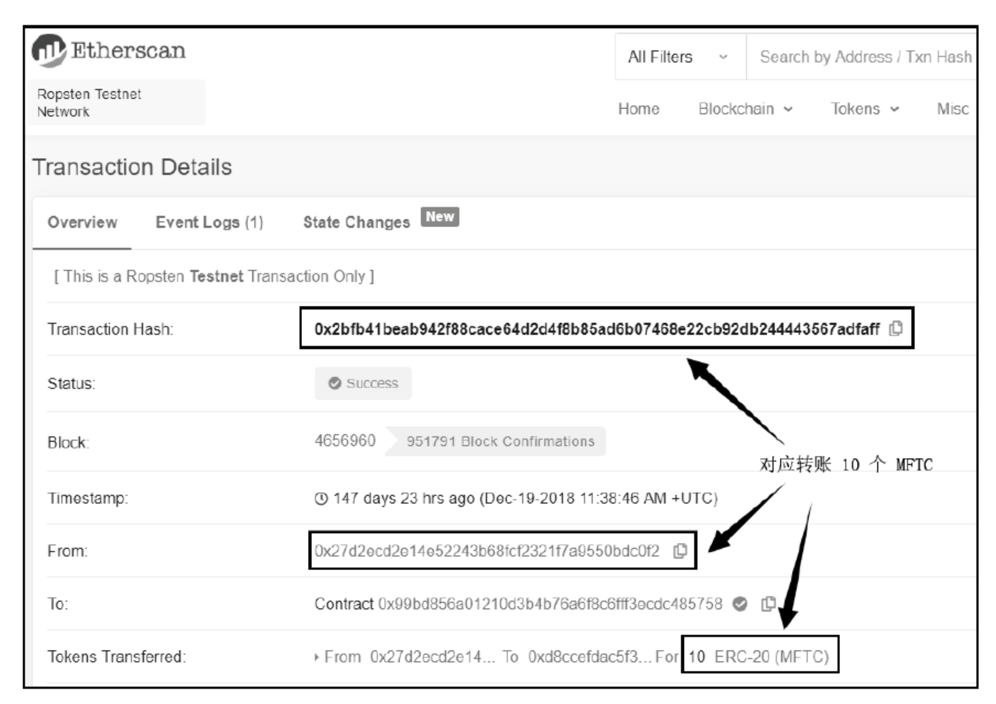



图 5-36　转账成功

## 5.3　区块事件监听

在本节中，我们将实现“以太坊中继”中最重要，也是最复杂的一部分功能，即通过遍历区块内部的交易记录来实现区块事件的监听，其必要性以及实现原理见“区块遍历”一节。

代码的实现步骤如下：

（1）从数据库中获取上一次成功遍历的非分叉状态的区块信息得到区块号 A，或通过以太坊接口“eth_blockNumber”获取新生成区块的区块号 A。

（2）调用以太坊接口“eth_blockNumber”，获取最新生成区块的区块号 B。

（3）比较 A 和 B 的大小关系，得出目标区块号“target”。

（4）得到“target”后，调用以太坊接口“eth_getBlockByNumber”获取区块的数据。

（5）数据库保存“target”对应的区块信息。

（6）检测是否存在区块分叉，这个步骤可以得出分叉事件。

（7）解析区块内的数据，读取内部的“transactions”交易信息，分析得出各种合约事件。

（8）数据库保存每笔交易信息。

从上面的步骤可知，我们需要用到数据库。数据库中的表有两个，一个是存储区块信息的表；另一个是存储区块内交易信息的表。所涉及的以太坊接口调用有 3 个：一个是“eth_blockNumber”；另一个是“eth_getBlockByNumber”；最后一个是“eth_getBlockByHash”，该接口在检测区块分叉的时候会用到。这 3 个接口的访问代码请参考“编写访问接口代码”一节。

对应上述步骤，我们给出遍历区块实现事件监听的流程图，如图 5-37 所示。

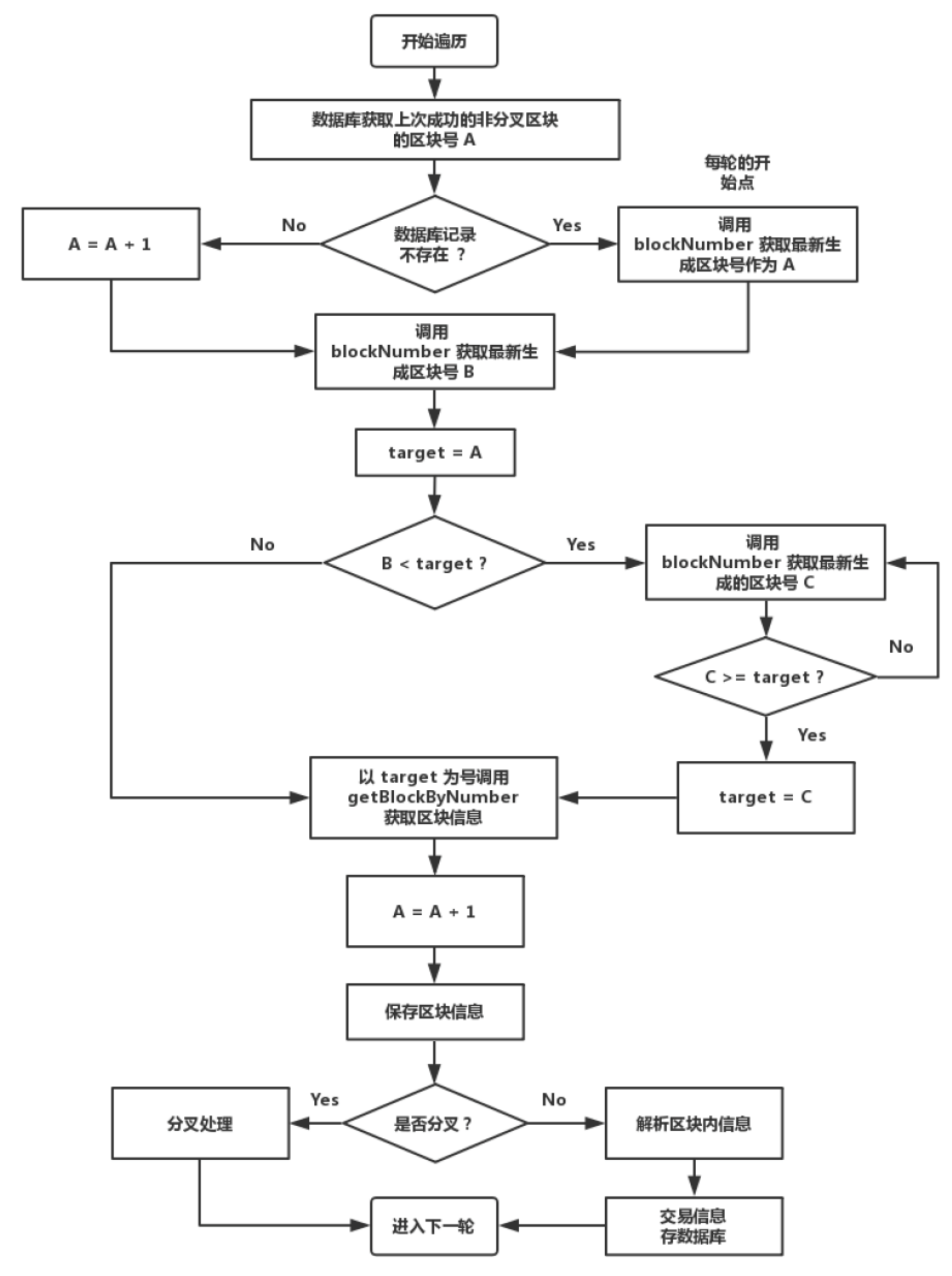



图 5-37　遍历区块实现事件监听的流程图

完成上面的整体遍历代码比较多，下面将分几个小节进行详细讲解。

## 5.3.1　创建数据库

区块遍历的存储采用 MySQL 数据库，在安装好了 MySQL 并启动之后，进入到数据库控制台创建数据库，以供以太坊中继程序使用，可以参考下面的 SQL 语句创建一个名为 eth_relay、字符集编码为 utf8 的数据库，使用 utf8 字符集编码是因为 utf8 字符集可以让数据库中的表兼容中文。

在控制台输入下面的代码并按回车键：

CREATE DATABASE eth_relay DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;

创建完数据库后，输入下面的代码，可看到刚刚创建的数据库（见图 5-38）：

```
Show databases;
```

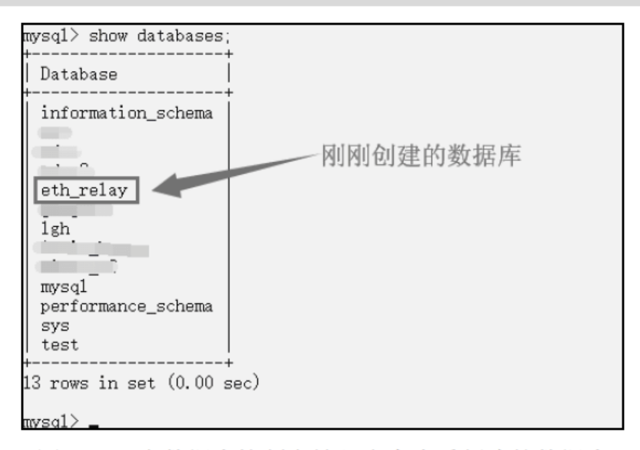



图 5-38　在数据库控制台输入命令查看创建的数据库

## 5.3.2　实现数据库的连接器

在创建完数据库后，我们需要在代码中完成一个专门用来管理数据库连接的对象。

首先在项目主文件夹下创建一个“dao”文件夹，专门用来存放数据库相关的代码文件，再到该文件夹下创建一个名称为“mysql.go”的文件。接下来在这个文件中编写数据库连接器相关的代码，如图 5-39 所示。




图 5-39　创建一个名称为“mysql.go”的文件

要想在 Golang 语言中使用 MySQL 的功能，目前有两种选择：

（1）自己动手写一个 MySQL 操作库。

（2）使用第三方开源的支持 MySQL 的数据库操作库。

对于上面的两种选择，我们采用第二种，学有余力的开发者也可以尝试自己编写数据库操作库。目前第三方开源的数据库操作库有很多，例如 gorm、xorm 等，在下面的示例中，我们选择“xorm”作为项目的数据库操作库。“xorm”库的官方技术文档链接是 http://www.xorm.io/docs/。

在 Goland 界面的底部打开“Terminal”控制台，然后使用 Golang 的“go get”命令下载远程

依赖包。我们需要下载两个，对应的命令（见图 5-40）分别是：

（1）go get github.com/go-sql-driver/mysql（下载“MySQL”操作库）。

（2）go get github.com/go-xorm/xorm（下载“xorm”数据库操作库）。

上面之所以还要下载“go-sql-driver/mysql”库，是因为“xorm”内部的代码中依赖到它，这也是代码库与代码库之间互相依赖要做的常规操作。

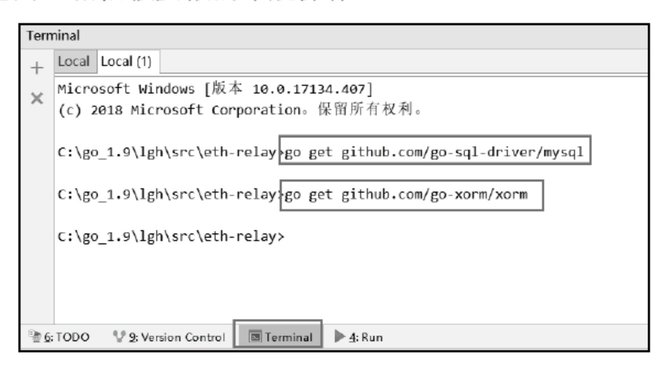



图 5-40　使用 Go 语言的 Get 命令获取依赖包

### 1.定义连接器

MySQL 数据库的连接需要通过一系列参数来完成，必需的参数有下面 5 个：

- 数据库域名，就是数据所在计算机的“IP”地址。
- 数据库端口，数据库程序在启动成功后监听电脑的应用程序端口，默认的端口是 3306。
- 数据库名称，程序连接数据库软件的时候所要使用的数据库名称，例如我们前面一节所创建的“eth_relay”。
- 数据库用户，就是在安装数据库时设置的用户，数据库登录用户名默认是“root”。
- 数据库密码，对应登录用户名的登录密码。

除了上面的参数外，数据库设置方面还有其他的参数可以选择。为了方便管理，我们在“mysql.go”文件中定义了一个 MySQL 配置信息结构体，代码如下所示：

```
// MySQL 连接配置信息
type MysqlOptions struct {
  Hostname           string  // 数据库服务器域名
  Port               string  // 端口
  User               string  // 数据库用户
  Password           string  // 数据库密码
  DbName             string  // 数据库名称
  TablePrefix        string  // 数据库表前缀
  MaxOpenConnections int     // 数据库最大连接数
  MaxIdleConnections int     // 数据库最大空闲连接数
  ConnMaxLifetime    int     // 空闲连接多长时间被回收，单位为秒
}
```

连接的配置信息是连接器所拥有的属性，因此我们可以在定义连接器结构体的时候将配置信息结构体添加到里面，作为它的一项属性。此外，最为重要的是连接器中需要拥有“xorm”框架的实例，这样才能使用“xorm”来操作数据库。最终连接器的结构体代码如下：

```
// MySQL 连接器结构体
type MySQLConnector struct {
   options *MysqlOptions    // 数据库配置结构体指针
   tables  []interface{}   // 数据库表的结构体集合
   Db      *xorm.Engine    // xorm 框架指针
}
```

### 2. 连接数据库

在定义好了连接器后，现在来实现数据库的连接。实现连接的代码将会编写在连接器初始化的函数内，由于第三方框架封装好了数据库连接及增、删、改、查的数据库操作，使得我们在使用这些数据库功能的时候不需要编写很多代码。

初始化函数的整体逻辑分为 4 步：

（1）连接数据库。

（2）设置数据库配置。

（3）不存在则创建数据表。

（4）同步表格的结构变化。

整体代码如下所示，“NewMqSQLConnector”函数负责实例化数据库连接器，“createTables”负责创建和同步数据表。在“xorm”中，当数据库域名与本地计算机相关时，数据库连接可以省略域名与端口设置。

```
// tables 是数据表的结构体实例数组
func NewMqSQLConnector(options *MysqlOptions, tables []interface{})
MySQLConnector {
   var connector MySQLConnector
   connector.options = options
   connector.tables = tables
   // 设置数据库连接的 url
   url := ""
   if options.Hostname == "" || options.Hostname == "127.0.0.1" {
      url = fmt.Sprintf(
         "%s:%s@/%s?charset=utf8&parseTime=True",
         options.User, options.Password, options.DbName)
   } else {
      url = fmt.Sprintf(
         "%s:%s@tcp(%s:%s)/%s?charset=utf8&parseTime=True",
         options.User, options.Password, options.Hostname, options.Port,
options.DbName)
   }
   db, err := xorm.NewEngine("mysql", url) // 以 MySQL 数据库类型实例化
   if err != nil {
      panic(fmt.Errorf("数据库初始化失败 %s", err.Error()))
   }
   tbMapper := core.NewPrefixMapper(core.SnakeMapper{}, options.TablePrefix)
   db.SetTableMapper(tbMapper)
   db.DB().SetConnMaxLifetime(time.Duration(options.ConnMaxLifetime) *
time.Second)
   db.DB().SetMaxIdleConns(options.MaxIdleConnections)
   db.DB().SetMaxOpenConns(options.MaxOpenConnections)
   // db.ShowSQL(true) // 是否开启打印 SQL 日志到控制台
```

```
  if err = db.Ping();err != nil {
    panic(fmt.Errorf("数据库连接失败 %s", err.Error()))
  }
  connector.Db = db
  // 创建数据表，策略是不存在则创建
  if err := connector.createTables(); err != nil {
    panic(fmt.Errorf("创建数据表失败 %s", err.Error()))
  }
  return connector
}

// 创建数据表，策略是不存在则创建
func (s *MySQLConnector) createTables() error {
  if len(s.tables) == 0 {
    // 没有数据表则需要创建
    return nil
  }
  if err := s.Db.CreateTables(s.tables...); err != nil {
    return fmt.Errorf("create mysql table error:%s", err.Error())
  }
  // 同步数据表的修改
  if err := s.Db.Sync2(s.tables...); err != nil {
    return fmt.Errorf("sync table error:%s", err.Error())
  }
  return nil
}
```

在“dao”文件夹下新建一个 MySQL 单元测试文件“mysql_test.go”，编写单元测试代码如下：

```
// 测试连接数据库
func Test_NewMqSQLConnector(t *testing.T) {
  option := MysqlOptions{
    Hostname:          "127.0.0.1",  // 本地数据库
    Port:             "3306",        // 默认端口
    DbName:            "eth_relay",   // 数据库名称
    User:             "root",        // 用户名
    Password:          "123456",     // 密码
    TablePrefix:        "eth_",       // 数据表前缀
    MaxOpenConnections: 10,
    MaxIdleConnections: 5,
    ConnMaxLifetime:    15,
  }
  tables := []interface{}{} // 不创建数据表
  mysql := NewMqSQLConnector(&option, tables)
  if mysql.Db.Ping() == nil {
    fmt.Println("数据库连接成功")
  }else{
    fmt.Println("数据库连接失败")
  }
}
```

测试结果如图 5-41 所示，可以看到数据库连接成功了。

```
Run:  Test_NewMqSQLConnector in eth-relay/dao
  Tests passed: 1 of 1 test – 21 ms
<4 go setup calls>
[xorm] [info]  2018/12/12 20:21:57.747269 PING DATABASE mysql
[xorm] [info]  2018/12/12 20:21:57.769210 PING DATABASE mysql
数据库连接成功

Process finished with exit code 0
```

图 5-41 连接数据库单元测试函数的运行结果

## 5.3.3 生成数据表

在 MySQL 中创建数据表的方式有两种：第一种是直接在 MySQL 的控制台中输入创建数据表的 SQL 语句进行创建；另一种是在代码中使用数据库 ORM 框架进行创建。第二种方式的操作往往比较方便。本项目使用的“xorm”数据库操作库就支持第二种方式。

### 1. 定义数据表

根据“xorm”文档的介绍，我们可以在代码文件中以定义数据结构体的形式来对应要创建的数据表。如下代码所示为存储区块信息数据表的数据结构体。需要注意的是，目前定义的结构体不需要存储完以太坊区块的所有数据字段，只挑选重要的部分进行存储即可。

```
// 存储区块信息的区块结构体
type Block struct {
   Id           int64   `json:"id"`              // 主键
   BlockNumber string  `json:"block_number"`  // 区块号
   BlockHash    string  `json:"block_hash"`    // 区块的哈希值
   ParentHash   string  `json:"parent_hash"`   // 父区块的哈希值
   CreateTime   int64   `json:"create_time"`   // 区块的生成时间
   Fork         bool    `json:"fork"`            // 是否为分叉区块
}
```

区块结构体的代码编写在“dao”文件夹下的新建文件“block.go”中，如图 5-42 所示。

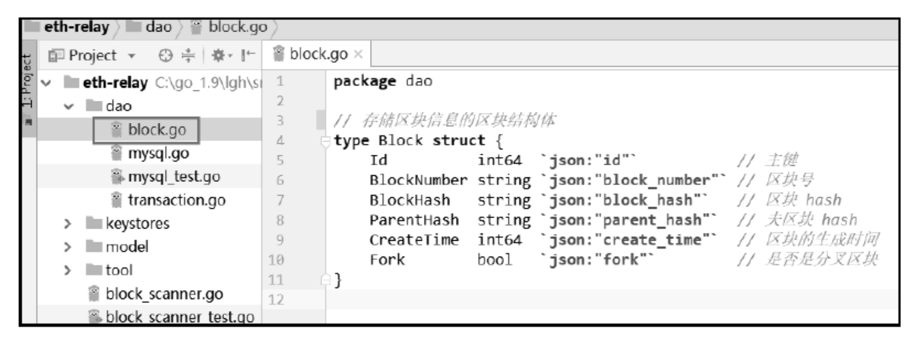



图 5-42 区块结构体代码

同样地，在“dao”文件夹下创建“transaction.go”文件，用来编写代码以便从存储区块中解析出交易信息结构体。代码如下所示：

```
// 对应于数据库表的交易数据结构体
type Transaction struct {
```

```
  Id             int64  `json:"id"`                  // 主键
  Hash           string `json:"hash"`                // 交易的哈希值
  Nonce          string `json:"nonce"`               // 交易的序列号
  BlockHash      string `json:"blockHash"`           // 当前交易被打包的区块的哈希值
  BlockNumber    string `json:"blockNumber"`         // 当前交易被打包在的区块的区块号
  TransactionIndex string `json:"transactionIndex"`
// 当前交易在区块已打包交易数组中的下标
  From           string `json:"from"`                // 交易发起者的地址
  To             string `json:"to"`                  // 交易接收者的地址
  Value          string `json:"value"`               // 交易的数值
  GasPrice       string `json:"gasPrice"`            // gasPrice
  Gas            string `json:"gas"`                 // gasLimit
  Input          string `xorm:"text" json:"input"`   // data
}
```

其中，“input”变量在数据表中的数据类型是“text”。“text”在 MySQL 中是文本存储类型，因为“input”的内容比较多，所以将该变量设为“text”类型，以存储较多的数据。根据“xorm”技术文档的介绍，如果在 Go 代码中的变量使用默认的“string”类型，而不在创建数据表的时候指定类型，那么默认将会使用“varchar(255)”类型，这种类型的变量是不能存储过多数据的，此时如果在进行插入操作的时候超出了数据量，就会出现数据库错误。

### 2. 创建数据表

使用“xorm”库来创建数据表是比较简单的，直接调用我们前面在数据库连接器代码中实现的“NewMqSQLConnector”函数，对定义好的“Block”和“Transaction”结构体传入参数即可。

修改之前的连接数据库的测试函数，添加创建表格的代码，让测试函数在连接数据库成功后进行数据表的创建，测试代码如下：

```
// 测试连接数据库，同时创建数据表
func Test_NewMqSQLConnector(t *testing.T) {
  option := MysqlOptions{
    Hostname:           "127.0.0.1",   // 本地数据库
    Port:               "3306",        // 默认端口
    DbName:             "eth_relay",   // 数据库名称
    User:               "root",        // 用户名
    Password:           "123456",      // 密码
    TablePrefix:        "eth_",        // 数据表前缀
    MaxOpenConnections: 10,
    MaxIdleConnections: 5,
    ConnMaxLifetime:    15,
  }
  tables := []interface{}{}
  tables = append(tables,Block{},Transaction{}) // 添加数据表的数据结构体
  NewMqSQLConnector(&option, tables)
// 传参进去，对应的结构体将会被 xorm 自动解析并创建数据表
  fmt.Println("创建数据表成功")
}
```

“xorm”根据数据结构体创建数据表名称的方式是，在所设置的数据表前缀“TablePrefix”上加上“_”，再加上结构体名称的小写字母。如果没有设置数据表前缀，那么将会直接使用结构体名称的小写，例如“Block”数据表在数据库里面对应的数据表名称是 eth_block。

运行上述的测试函数后，可以在 MySQL 数据库控制台输入如下命令组来查看数据表是否生成：

```
use eth_relay;
show tables;
```

测试与检验结果如图 5-43 所示。




```
mysql> use eth_relay;
Database changed
mysql> show tables;
+-------------------+
| Tables_in_eth_relay |
+-------------------+
| eth_block         |
| eth_transaction   |
+-------------------+
2 rows in set (0.00 sec)

mysql>
```

数据表创建成功

图 5-43　在数据库控制台输入命令查看创建好的数据表

## 5.3.4　区块遍历器

创建好数据库和数据表之后，在项目文件夹“eth-relay”下创建文件“block_scanner.go”，如图 5-44 所示。

```
eth-relay > block_scanner.go
Project
eth-relay C:\go_1.9\lgh\src\eth-r
  dao
  keystores
  model
  tool
  block_scanner.go
  block_scanner_test.go
  ETH_RPC_Client.go
  ETH_RPC_Requester.go
  ethrpc_test.go
  main.go
  nonce_manager.go

block_scanner.go
1  package main
```

图 5-44　创建的“block_scanner.go”　文件

区块扫描器的代码将编写在该文件中，由于扫描器的实现代码比较多，因此下面将按照功能进行分模块讲解。

### 1. 定义遍历器

因为要通过访问以太坊接口来获取区块内的数据，所以区块遍历器结构体中需要定义一个以太坊的“RPC”请求者，即我们前面内容中所编写的“ETHRPCRequester”。此外，区块遍历后获取的数据要存放到数据库中，那么遍历器结构体中还需要定义数据库连接器对象。

区块遍历是一个循环过程，为达到区块分叉检测，需要在每次成功遍历后，在内存中存储上一次遍历的区块，以便在新一轮的遍历中把当前轮次区块的哈希值与上次的哈希值进行比较，判断它们是否一致，如果不一致，就证明出现了分叉。因此在遍历器的结构体中还需要定义用来存储每次遍历成功后上一次的区块。

区块遍历器的结构体定义及其实例化代码如下所示：

```
// 区块遍历器
type BlockScanner struct {
   ethRequester ETHRPCRequester    // 以太坊 rpc 请求者对象
   mysql        dao.MySQLConnector  // 数据库连接器对象
   lastBlock     *dao.Block         // 用来存储每次遍历后上一次的区块
   lastNumber    *big.Int           // 上一次区块的区块号
   fork         bool               // 区块分叉标记位
   stop         chan bool          // 用来控制是否停止遍历的管道
   lock         sync.Mutex         // 互斥锁，控制并发
}

func NewBlockScanner(requester ETHRPCRequester, mysql dao.MySQLConnector)
*BlockScanner {
   return &BlockScanner{
      ethRequester: requester,
      mysql:        mysql,
      lastBlock:    &dao.Block{},
      fork:         false,
      stop:         make(chan bool),
      lock:         sync.Mutex{},
   }
}
```

### 2. 区块分叉检测

在以太坊中，分叉区块中打包了的交易是不算数的，也就是无效的，所以我们在遍历区块时要过滤掉分叉区块，对这些区块不做交易读取处理。

以太坊区块分叉在代码层面主要是通过区块父子关系的哈希值进行判断。举个例子，在区块高度为 15 的时候，我们获取到区块 A 的哈希值是“0x123”，此时高度累加 1 变为 16，我们根据 16 的高度去获取对应的区块 B，然后判断区块 B 的父块哈希值（parent hash）是否是区块 A 的哈希值。因为高度 15 的区块必须是高度 16 区块的父区块，所以 A 区块的哈希值必须要等于 B 区块的父块哈希值，否则就是分叉了。

根据上述区块分叉的判断条件结合“区块遍历器”结构体中上一次的区块变量，我们可以编写如下判断是否分叉的代码：

```
// 判断是否分叉的函数，若返回 true，则是分叉
func (scanner *BlockScanner) isFork(currentBlock *dao.Block) bool {
   if currentBlock.BlockNumber == "" {
      panic("invalid block")
   }
   // scanner.lastBlock.BlockHash == currentBlock.ParentHash 判断上一次的区块哈希值
是否是当前区块的父区块哈希值
   if scanner.lastBlock.BlockHash == currentBlock.BlockHash ||
scanner.lastBlock.BlockHash == currentBlock.ParentHash {
      scanner.lastBlock = currentBlock // 没有发生分叉，更新上一次区块为当前被检测的区块
      return false
   }
   return true
}
```

### 3. 获取分叉点区块的思路

分叉点区块的意思是某个区块分叉是从它开始的。在检测出存在分叉区块后，需要在数据库中找到当前分叉区块的“分叉点区块”。然后将从该“分叉点区块”的区块号开始到分叉块区块号之间的区块全部标记为分叉，标志位对应“block”区块结构体中的“fork”变量，如图 5-45 所示。

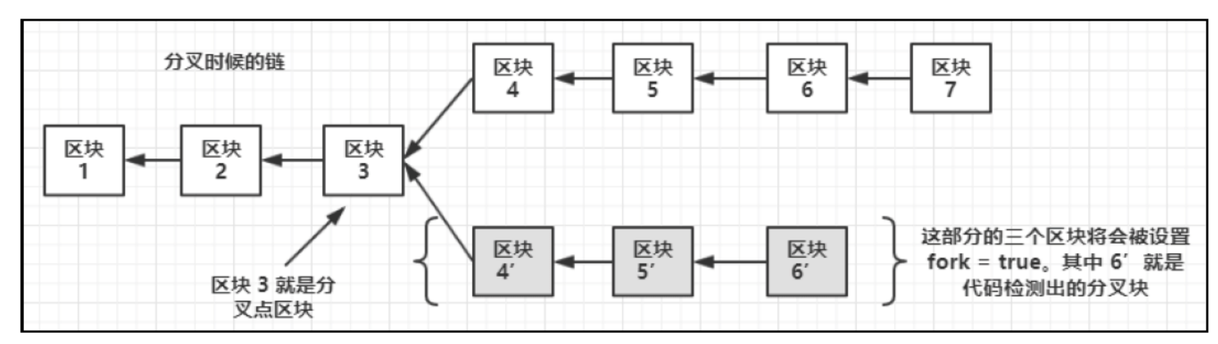



图 5-45 分叉点区块

寻找“分叉点区块”需要理解下面的知识点：

- 寻找所依赖的是“父块哈希值”，即父区块的哈希值。
- 寻找算法是递归算法，不断地递归直至找到分叉区块之前在本地数据库中存在的区块时，再跳出递归。
- 寻找区块的步骤是先从本地的数据库中寻找，因为我们在每次成功获取一个区块的信息后都会把它存储于本地的数据库中，如果本地数据库没有寻找到，就再访问以太坊接口“eth_getBlockByHash”来获取。

整体的流程图如图 5-46 所示。

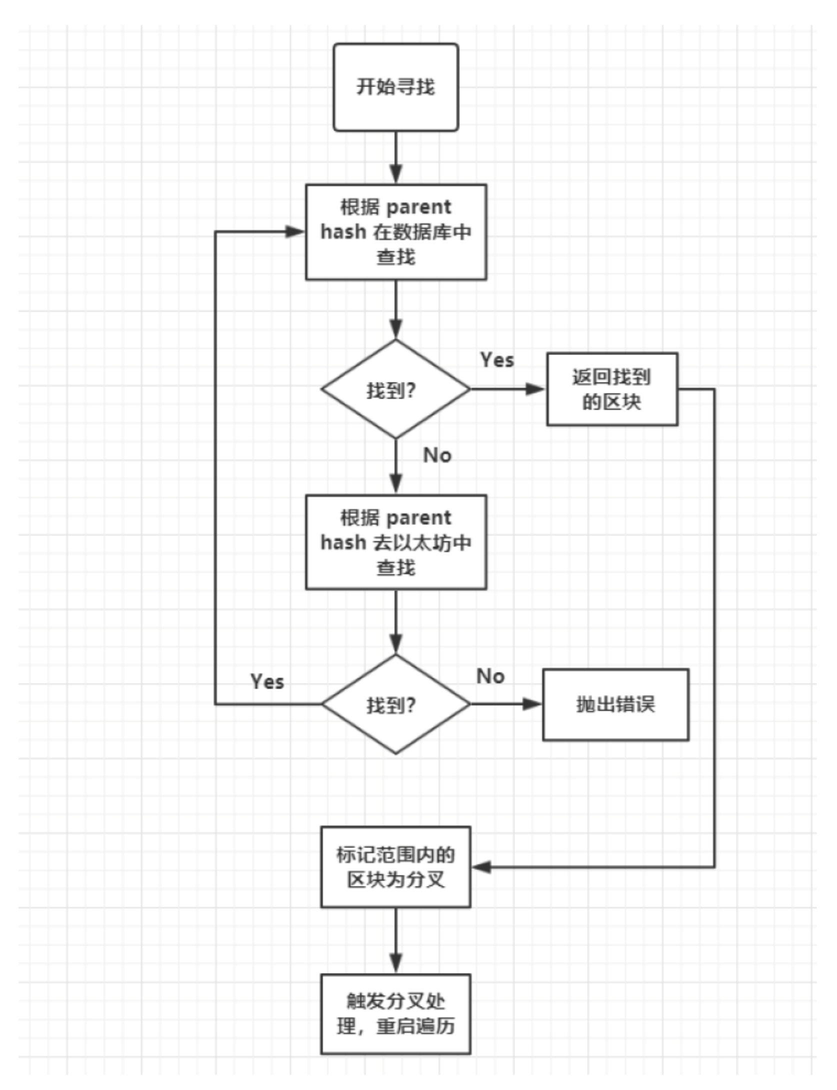



图 5-46　寻找分叉点区块的设计流程图

综合流程图和上面的“检测分叉图”，下面我们再通过图文进行理解。

假设此时区块遍历没有出现分叉的情况，正常地进行到了 3A 区块，如图 5-47 所示。

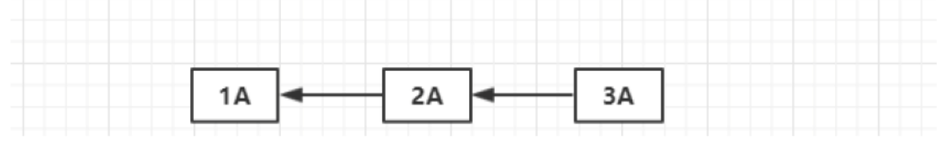



图 5-47　正常遍历到区块 3A

区块遍历继续进行，此时遍历到 5A，发现没有出现分叉，然后将遍历器中的“lastBlock”变量设置为 5A，在下一次的循环中将继续检测分叉，如图 5-48 所示。

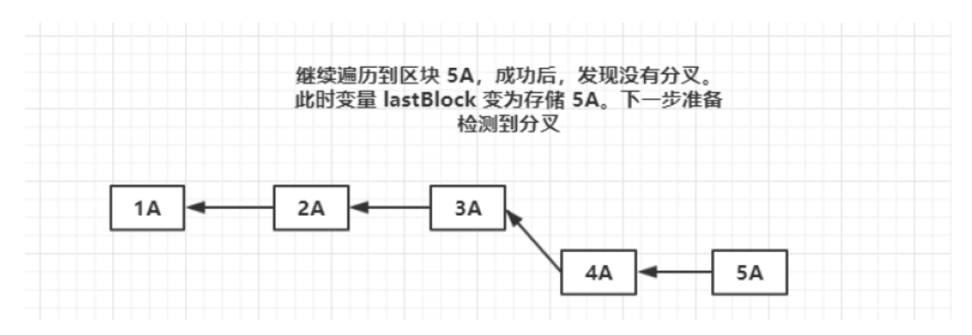



图 5-48 区块正常遍历

当获取到以太坊最新生成（latest）的区块时，此时节点中刚好同步好了一条最优链（4B，5B，6B 所在链），最优链中，6B 区块是最新的，因此节点便将 6B 区块返回给客户端。客户端一旦发现 6B 区块的父块哈希值（parent hash）和“lastBlock”的哈希值不相等，就证明检测到了分叉。此时到本地数据库查找分叉点区块，由于 5B 和 4B 区块还没有存储到本地，自然就不可能获取到，这样在查找的过程中只能通过访问以太坊的 “eth_getBlockByHash” 接口来获取 4B 和 5B 区块。最终在获取到 4B 区块的时候，根据 4B 区块的父块哈希值（parent hash）从数据库中获取到了 3A 区块，3A 区块就是分叉点区块，然后程序将 3A 区块到 6B 区块中间的 4A 和 5A 区块都标记为分叉块。最后程序进行遍历重启，从 3A 区块开始，补充完 4B 和 5B 区块。整个遍历过程如图 5-49 所示。

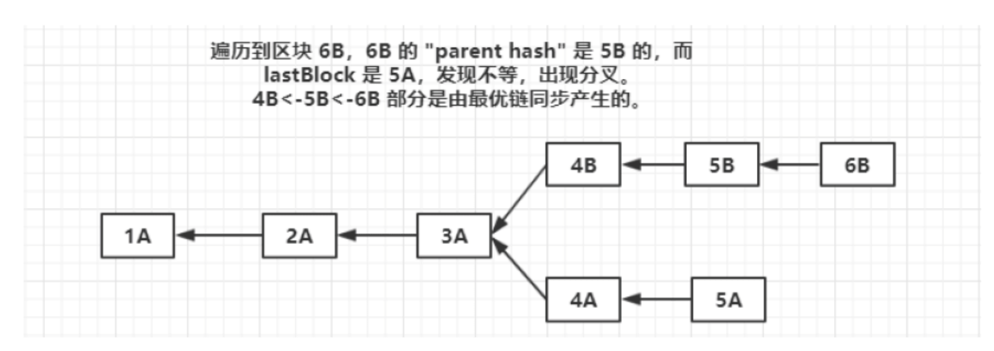



图 5-49 遍历出现分叉情况

### 4. 编写获取分叉点区块的代码

根据前面的思路分析，我们可以给出实现获取分叉点区块代码的思路：

（1）根据检测出的分叉块的父块哈希值（parent hash）进行本地数据库的查找，若存在则直接返回。

（2）如果本地数据库不存在，那么调用请求以太坊接口获取父区块（简称为父块）信息。

（3）在获取到了父区块后，继续往上递归查看该父区块的父区块，直到在本地数据库找到分叉点区块。

```
// 获取分叉点区块
func (scanner *BlockScanner) getStartForkBlock(parentHash string) (*dao.Block,
error) {
  // 获取当前区块的父区块，分叉从父区块开始
  parent := dao.Block{} // 定义一个 block 结构体实例，用来存储从数据库查询出的区块信息
  // 下面使用 xorm 框架提供的函数，根据 block_hash 去数据库获取区块信息，等同于 SQL 语句：
```

```
    // select * from eth_block where block_hash=parentHash limit 1;
    _, err := scanner.mysql.Db.Where("block_hash=?", parentHash).Get(&parent)
    if err == nil && parent.BlockNumber != "" {
        return &parent, nil  // 本地存在，直接返回分叉点区块
    }
    // 数据库没有父区块记录，准备从以太坊接口获取
    parentFull, err := scanner.retryGetBlockInfoByHash(parentHash)
    if err != nil {
        return nil, fmt.Errorf("分叉严重错误，需要重启区块扫描 %s", err.Error())
    }
    // 继续递归往上查询，直到在数据库中有它的记录
    return scanner.getStartForkBlock(parentFull.ParentHash)
}

// 输出日志
func (scanner *BlockScanner) log(args ...interface{}) {
    fmt.Println(args...)
}
```

其中，“retryGetBlockInfoByHash”函数是一个带有重试策略的“eth_getBlockByHash”的改版函数，之所以带重试策略，是为了防止因为网络或节点原因导致一次获取出错而使整个程序被中止。对于远程服务导致的错误，可给予请求重试。图 5-50 就是由于远程服务错误后重试获取成功的截图。




图 5-50　区块信息获取的重试策略

重试策略函数有两个，包括根据区块号获取信息的函数以及根据区块哈希值获取信息的函数。它们的实现代码分别如下：

```
// 区块号存在，信息获取为空，可能是以太坊网络延时问题，重试策略函数
func (scanner *BlockScanner) retryGetBlockInfoByNumber(targetNumber *big.Int)
(*model.FullBlock, error) {
Retry:
    // 下面调用以太坊请求者 ethRequester 的 GetBlockInfoByNumber 函数
    fullBlock, err := scanner.ethRequester.GetBlockInfoByNumber(targetNumber)
    if err != nil {
        errInfo := err.Error()
        if strings.Contains(errInfo, "empty") {
            // 区块号存在，信息获取为空，可能是以太坊网络延时问题，直接重试
            scanner.log("获取区块信息，重试一次.....", targetNumber.String())
            goto Retry
        }
```

```
        return nil, err
    }
    return fullBlock, nil
}

// 区块哈希值存在，信息获取为空，可能是以太坊网络或节点问题，重试策略函数
func (scanner *BlockScanner) retryGetBlockInfoByHash(hash string)
(*model.FullBlock, error) {
Retry:
    // 下面调用我们以太坊请求者 ethRequester 的 GetBlockInfoByHash 函数
    fullBlock, err := scanner.ethRequester.GetBlockInfoByHash(hash)
    if err != nil {
        errInfo := err.Error()
        if strings.Contains(errInfo, "empty") {
            // 区块号存在，信息获取为空，可能是以太坊网络延时问题，直接重试
            scanner.log("获取区块信息，重试一次.....", hash)
            goto Retry
        }
        return nil, err
    }
    return fullBlock, nil
}
```

### 5. 获取要进行扫描的区块号

区块号在整个遍历流程中充当了数据请求的前置条件，需要根据不同的情况正确设置区块号的值。一般要考虑的情况有：

（1）程序首次启动时，应该如何赋值。

（2）程序第 N（N ≥ 1 时）次启动时区块号的取值。

（3）程序运行中，区块号的值应该如何变化。

结合前面“区块事件监听”的整体流程图可知，首先需要从数据库中查找出上一次成功遍历的且不是分叉的区块，然后判断是否存在区块的数据：如果区块有数据，那么对应上述的第（2）点，这是程序的第 N 次启动；如果区块没有数据，是空区块，那么证明程序是首次启动。

实现上述第（1）与（2）点的代码如下，请务必跟随代码中的注释进行理解：

```
// 初始化：内部在开始遍历时赋值 lastBlock
func (scanner *BlockScanner) init() error {
    // 下面使用 xorm 提供的数据库函数来从
    // 数据库中寻找出上一次成功遍历的且不是分叉的区块
    // 等同于 SQL：select * from eth_block where fork=false order by create_time desc
limit 1;
    _, err := scanner.mysql.Db.
        Desc("create_time"). // 根据时间降序
        Where("fork = ?", false).
        Get(scanner.lastBlock)
    if err != nil {
        return err
    }
    if scanner.lastBlock.BlockHash == "" {
        // 区块哈希值为空，证明是整个程序的首次启动，那么从节点中获取最新生成的区块
```

```
    // GetLatestBlockNumber 获取最新区块的区块号
    latestBlockNumber, err := scanner.ethRequester.GetLatestBlockNumber()
    if err != nil {
      return err
    }
    // GetBlockInfoByNumber 根据区块号获取区块数据
    latestBlock, err :=
scanner.ethRequester.GetBlockInfoByNumber(latestBlockNumber)
    if err != nil {
      return err
    }
    if latestBlock.Number == "" {
      panic(latestBlockNumber.String())
    }
    // 下面是给区块遍历器的 lastBlock 变量赋值
    scanner.lastBlock.BlockHash = latestBlock.Hash
    scanner.lastBlock.ParentHash = latestBlock.ParentHash
    scanner.lastBlock.BlockNumber = latestBlock.Number
    scanner.lastBlock.CreateTime =
scanner.hexToTen(latestBlock.Timestamp).Int64()
    scanner.lastNumber = latestBlockNumber
  } else {
    // 区块哈希值不为空，证明不是首次启动，而是后续的启动
    scanner.lastNumber, _ =
new(big.Int).SetString(scanner.lastBlock.BlockNumber, 10)
    // 下面加 1，因为上一次数据库存的是已经遍历完了的区块，接下来是它的下一个区块
    scanner.lastNumber.Add(scanner.lastNumber, new(big.Int).SetInt64(1))
  }
  return nil
}

// 定义一个将十六进制数转为十进制大数的函数
func (scanner *BlockScanner) hexToTen(hex string) *big.Int {
  if !strings.HasPrefix(hex,"0x") {
    ten, _ := new(big.Int).SetString(hex, 10) // 本身就是十进制字符串，直接设置
    return ten
  }
  ten, _ := new(big.Int).SetString(hex[2:], 16)
  return ten
}
```

第（3）点的情况，需要每次和最新区块号进行比较。在代码中首先要获取公链上当前最新生成区块的区块号，假设它是 A，然后使用 A 和在初始化时设置“lastBlock”中的区块号 B 进行比较，可能会出现的情况有：

- A<B，说明 B 过大，此时要循环获取最新的 A，直到 A ⩾ B 才开始遍历。一般来说，这种情况很少出现，除非特定地设置要从某个高度开始遍历。例如，当前最新区块高度是 5，那么故意设置从 8 高度才开始遍历就会出现这种情况。
- A ⩾ B，证明 B 恰好等于当前的最新区块高度或者比最新区块高度要小，可以继续从 B 开始遍历区块。

第 3 点的代码实现如下：

```
// 获取要扫描的区块号
func (scanner *BlockScanner) getScannerBlockNumber() (*big.Int,error) {
   // 调用以太坊请求者 ethRequester 获取公链上最新生成的区块的区块号
   newBlockNumber, err := scanner.ethRequester.GetLatestBlockNumber()
   if err != nil {
      return nil,err
   }
   latestNumber := newBlockNumber
   // 下面使用 new 的形式初始化并设置值，不要直接赋值，
   // 否则会和 lastNumber 的内存地址一样，影响后面的获取区块信息
   targetNumber := new(big.Int).Set(scanner.lastNumber)
   // 比较区块号大小
   // -1 if x <  y, 0 if x == y, +1 if x >  y
   if latestNumber.Cmp(scanner.lastNumber) < 0 {
      // 最新的区块高度比设置的要小，则等待新区块高度 >= 设置的
      Next:
         for {
            select {
            case <-time.After(time.Duration(4 * time.Second)): // 延时 4 秒重新获取
               number, err := scanner.ethRequester.GetLatestBlockNumber()
               if err == nil && number.Cmp(scanner.lastNumber) >= 0 {
                     break Next // 跳出循环
               }
            }
         }
   }
   return targetNumber,nil // 返回目标区块高度
}
```

需要注意的是，函数“init”的调用要比“getScannerBlockNumber”早，因为后者依赖前者设置好的“lastNumber”。

### 6. 实现区块扫描

扫描区块使用权函数就是上述我们实现每个功能的集合，其执行流程请参照“区块事件监听”的整体流程图。

扫描函数的完整代码如下所示：

```
// 扫描区块
func (scanner *BlockScanner) scan() error {
   // 获取要进行扫描的区块号
   targetNumber,err := scanner.getScannerBlockNumber()
   if err != nil {
      return err
   }
   // 使用具有重试策略的函数获取区块信息
   fullBlock, err := scanner.retryGetBlockInfoByNumber(targetNumber)
   if err != nil {
      return err
   }
   // 区块号加 1，在下次扫描的时候，指向下一个高度的区块
   scanner.lastNumber.Add(scanner.lastNumber, new(big.Int).SetInt64(1))
```

```
    // 因为涉及两张数据表的更新，我们需要采用数据库事务处理
    tx := scanner.mysql.Db.NewSession()  // 开启事务
    defer tx.Close()

    // 准备保存区块信息，先判断当前区块记录是否已经存在
    block := dao.Block{}
    _, err = tx.Where("block_hash=?", fullBlock.Hash).Get(&block)
    if err == nil && block.Id == 0 {
        // 不存在，进行添加
        block.BlockNumber = scanner.hexToTen(fullBlock.Number).String()
        block.ParentHash = fullBlock.ParentHash
        block.CreateTime = scanner.hexToTen(fullBlock.Timestamp).Int64()
        block.BlockHash = fullBlock.Hash
        block.Fork = false
        if _, err := tx.Insert(&block); err != nil {
            tx.Rollback() // 事务回滚
            return err
        }
    }
    // 检查区块是否分叉
    if scanner.forkCheck(&block) {
        data, _ := json.Marshal(fullBlock)
        scanner.log("分叉！", string(data))
        tx.Commit()  // 即使分叉了，也要把保存区块的事务提交
        scanner.fork = true // 发生分叉
        return errors.New("fork check")  // 返回错误，让上层处理并重启区块扫描
    }
    // 解析区块内的数据，读取内部的“transactions”交易信息，分析得出各种合约事件
    scanner.log(
          "scan block start ==> ", "number: ",
          scanner.hexToTen(fullBlock.Number), "hash: ", fullBlock.Hash)
    for index, transaction := range fullBlock.Transactions {
        // 下面的打印操作模拟自定义处理。对于每笔 tx，我们是完全可以进一步从里面提取信息的！
        scanner.log("tx hash ==> ", transaction.Hash)
        if index == 5 {
            // 因为每个区块打包的交易数目是不同的，为了减少显示的信息，这里控制只打印 5 笔
            break
        }
    }
    scanner.log("scan block finish \n=================")
    // 数据库保存交易信息
    if _, err = tx.Insert(&fullBlock.Transactions); err != nil {
        tx.Rollback()  // 事务回滚
        return err
    }
    return tx.Commit() // 提交事务
}
```

因为获取的区块数据要保存到数据库中，在区块遍历完成后，必须将所有交易记录的数据保存到数据库。这个过程涉及两张不同数据表的插入操作，为了防止一方插入失败而导致数据不对称，在代码中使用了 MySQL 事务操作进行数据插入。这样做的好处是在错误发生的时候可以及时地进行数据回滚。

### 7. 启动区块扫描

在完成了最后的区块扫描函数“scan”后，还需要一个启动整个扫描流程的函数，即启动函数“Start()”。

启动函数的执行步骤如下：

（1）首先为互斥锁上锁，防止多协程操作同一个“BlockScanner”实例去启动多次“Start”函数扫描。

（2）进行前置数据的初始化，例如获取上一次遍历成功的非分叉区块的区块号。

（3）启动“Go”协程。在内部调用“Scan”扫描函数，因为扫描动作不能阻塞在 main 函数的主协程中。

（4）在进行扫描的同时，还要监听“stop”管道，以捕获停止扫描的动作指示，以及在检测到分叉事件的时候，重新进行初始化前置数据，随后继续进行扫描。

“Start()”函数的完整代码如下：

```
// 整个区块扫描的启动函数
func (scanner *BlockScanner) Start() error {
   scanner.lock.Lock()  // 互斥锁加锁，在 stop 函数内有解锁步骤
   // 首先调用 init 进行数据初始化，内部主要是初始化区块号
   if err := scanner.init(); err != nil {
      scanner.lock.Unlock() // 因为出现了错误，所以我们要进行解锁
      return err
   }
   execute := func() {
      // scan 函数，就是区块扫描函数
      if err := scanner.scan(); nil != err {
         scanner.log(err.Error())
         return
      }
      time.Sleep(1 * time.Second) // 延迟一秒开始下一轮
   }
   // 启动一个 go 协程来遍历区块
   go func() {
      for {
         select {
         case <-scanner.stop: // 监听是否退出遍历
            scanner.log("finish block scanner!")
            return
         default:
            if !scanner.fork {
               // 进入这个 if 证明没有检测到分叉，正常地进行每一轮的遍历
               execute()
               continue
            }
            // 若 fork = true，则监听到有分叉，重新初始化
            // 重新从数据库获取上次遍历成功且没有分叉的区块号
            if err := scanner.init(); err != nil {
               scanner.log(err.Error())
               return
            }
            scanner.fork = false
```

```
        }
      }
   }()
   return nil
}

// 公有函数，可以供外部调用来停止区块遍历
func (scanner *BlockScanner) Stop() {
   scanner.lock.Unlock()  // 解锁
   scanner.stop <- true
}
```

至此，整个区块遍历器的代码都已经编写完成，全部编写在“block_scanner.go”文件中。接下来我们进行遍历器代码的测试。

### 8. 测试区块扫描器

关于区块扫描器的测试代码，我们在项目中单独新建一个 “block_scanner_test.go” 文件来编写，如图 5-51 所示。

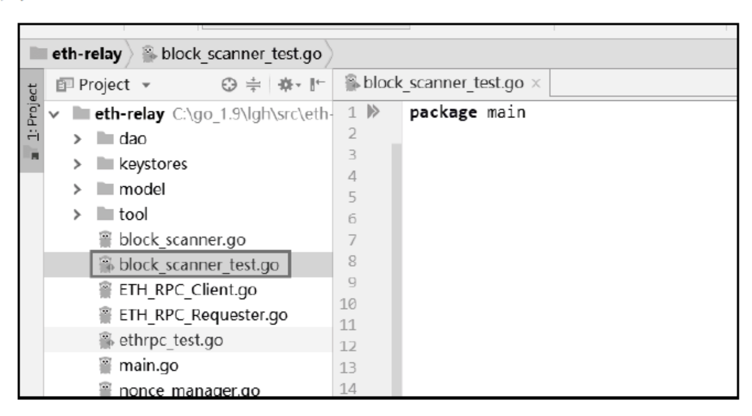



图 5-51　新建测试代码文件“block_scanner_test.go”

根据区块扫描器结构体的定义，测试代码中首先要定义好一个以太坊“RPC”请求者和数据库连接器，再根据它们去初始化区块扫描器。整体的测试代码如下：

```
// 单元测试：区块扫描器，开始扫描区块
func TestBlockScanner_Start(t *testing.T) {
   // 初始化以太坊 rpc 请求者
   mainNet := "https://mainnet.infura.io/v3/2e6d9331f74d472a9d47fe99f697ca2b"
   requester := NewETHRPCRequester(mainNet)

   // 初始化数据库连接器的配置对象，记得修改为本地数据库的参数
   option := dao.MysqlOptions{
      Hostname:          "127.0.0.1",
      Port:             "3306",
      DbName:           "eth_relay",
      User:             "root",
      Password:          "123456",
      TablePrefix:       "eth_",
      MaxOpenConnections: 10,
      MaxIdleConnections: 5,
      ConnMaxLifetime:    15,
```

```
    }
    // 添加数据表
    tables := []interface{}{}
    tables = append(tables, dao.Block{}, dao.Transaction{})

    // 根据上面定义的配置，初始化数据库连接器
    mysql := dao.NewMqSQLConnector(&option, tables)

    // 初始化区块扫描器
    scanner := NewBlockScanner(*requester, mysql)
    err := scanner.Start() // 开始扫描
    if err != nil {
        panic(err)
    }
    // 使用 select 模拟阻塞主协程，等待上面的代码执行，因为扫描是在 gorutine 协程中进行的
    select {}
}
```

运行单元测试，可以看到控制台开始输出遍历区块的数据，如图 5-52 所示。




图 5-52 测试区块扫描器时，控制台输出的日志

对应地，通过 Goland 底部的“Terminal”进入到 MySQL 数据库终端，查询“eth_block”数据表的数据，我们也能看到区块数据已经成功存储进去了（见图 5-53），查询命令是：

```
select * from eth_block;
```

mysql> select * from eth_block;

| id | block_number | block_hash | pare |
|---|---|---|---|
| 225 | 6926560 | 0xd23b42d6bcbb462f403e631684478ce8bcdd437fbb790c58ff66a315ebb13fad | 0x49 |
| 226 | 6926561 | 0x1c7876e7b4d1be07919db4a3c882441d7413b93cec35a7d10f6454c46d798a10 | 0xd2 |
| 227 | 6926562 | 0x019f9e23fbcbd07cfd255ab06e06b6d783a4bd19e745856e99df39a7d49f76e3 | 0x1c |
| 228 | 6926563 | 0x88a3d7577ac72063e233b61d6b5c25df883c98b4133b0515d00c21d27dbe9f83 | 0x01 |
| 229 | 6926564 | 0xc392bd1f526a9ae1b4d7174c9f0014a45e6ac151cc981751a7c3f350b84a2741 | 0x88 |
| 230 | 6926565 | 0x997e704f5e4ada136ebbf88e46f8b506c209792cc3829a0b90d7ec92cce5ad1c | 0xc3 |
| 231 | 6926566 | 0x9c323c9924466843fd060740bf4c2a25bd79bf47b1e8de754dd200e1385b1b5f | 0x99 |
| 232 | 6926567 | 0x9f3b65d58c3ccbd74918a39b49a810f9c53f2918689b838bf36f155aecd97c01 | 0x9c |
| 233 | 6926568 | 0xf4710ef1f6eba72c05f9a16b51cc5113bcc9d76709be17994017e120800a797c | 0x9f |
| 234 | 6926569 | 0xb7996973e39c89c3353d0b3229b6de1074a5e9c931594fe3d47b3caffce374c7 | 0xf4 |
| 235 | 6926570 | 0x2cbe1806d462bc60a9821b2dd16cb7b72ef3f5202b49a1295c3b84a2b056c600 | 0xb7 |
| 236 | 6926571 | 0x0532afedebf62e6e1500fd5daefc9163c4c5d9291899fb477c3e590f4aa5ee54 | 0x2c |
| 237 | 6926572 | 0x46e45e38db0620cd984114a01045176bfa953f0f7ce39e8b728b0e6131aeef56 | 0xcc |
| 238 | 6926571 | 0xcc1ba15d8062191d7ae3f312891958927f24d3828ab376d73e076409dd3f79d7 | 0x2c |
| 239 | 6926573 | 0x4aa2a705ef343bd93aa8c30009a71c68dbf0d660f99179c1afdaab0a94fb8584 | 0x46 |
| 240 | 6926574 | 0x01062731c1ea4447e9f8366d79fc23797538aace2c9af17612f6257b4a37bba6 | 0x4a |

图 5-53 在数据库控制台输入命令查看数据表中被遍历了的区块信息

继续观察程序的运行，一段时间后，就会看到检测到的分叉区块的日志输出。如图 5-54 所示，“69b0ec”的十进制是“6926572”，在读取“6926572”号区块的时候检测到分叉，然后往上递归找到分叉点区块是“6926570”，那么在“6926570”到“6926572”范围内的区块就是分叉区块，即“6926571”是分叉区块。此时程序跳出遍历，开始重新初始化，从数据库中获取出上次遍历成功的且没有分叉的区块“6926570”，然后“6926570”累加 1，为“6926571”，程序便从“6926571”继续开始往下遍历。

```
=================
scan block start ==> number: 6926571 hash: 0x0532afedebf62e6e1500fd5daefc9163c4c5d9291899fb477c3e590f4aa5ee54
tx hash ==>  0x65783382e687aad4113c39304ffd6efa9540ef6d112e91da60f1e3d817713006
tx hash ==>  0x676f3cfaac726254c4c9a50a1e1b79d64b4ee9d9dc6a71cec16696176b4bd077
tx hash ==>  0x3d8dffbb9ace05f0eae4f673f979d16351dd8c33f81845a3da25f15d1cb65bcb
tx hash ==>  0x83a8aa91d961c57d3e977a907e6581c8a249a2fbd12f700b5878b5105141a220
tx hash ==>  0x64dd1500fa6d342b53d5460021c30b5f065dc9d6fd2f7a45c12d0ccc542e98c4
tx hash ==>  0x9d549e3ca01212f524e9c9e04a91c52f0b82af04e77f09567e34b0fdd94f85d8
scan block finish
-----------------
分叉!  {"number":"0x69b0ec","hash":"0x46e45e38db0620cd984114a01045176bfa953f0f7ce39e8b728b0e6131aeef56","parentHa
fork check
scan block start ==> number: 6926571 hash: 0xcc1ba15d8062191d7ae3f312891958927f24d3828ab376d73e076409dd3f79d7
tx hash ==>  0x65783382e687aad4113c39304ffd6efa9540ef6d112e91da60f1e3d817713006
tx hash ==>  0x6188ac03f70376c6dcf647b18b37f4353f37b5ad28dddc3de5b2f3c69c974ffa
tx hash ==>  0x852fe2ee1472cdd86e3b65d42437556aed22bb0ddbf38cdc306a440ff3175a93
tx hash ==>  0x17a1c83e3853eeeeb7e57d02020dacc4d3eb1a99ec9914be0956f8571888276f
tx hash ==>  0x3b710c93ca3bc6b0fcdd5d0eaeaddfb9176cb85cbcef375535a1bef1106564de
tx hash ==>  0x20342f9e3df0389f7f2f625de95d6d2a5582c2b806ddd20e8e7015128a61aad0
scan block finish
=================
```

第一个 6926571 是分叉区块，它的hash 后字符是 54 结尾。重新扫描的是正常的区块，以 d7 字符结尾

图 5-54　区块遍历检测到分叉块后的重新纠正扫描所产生的日志

打开以太坊区块链浏览器，查询“6926571”号区块的哈希值是否是以“d7”结尾，以验证以“54”结尾的是分叉区块，如图 5-55 所示。

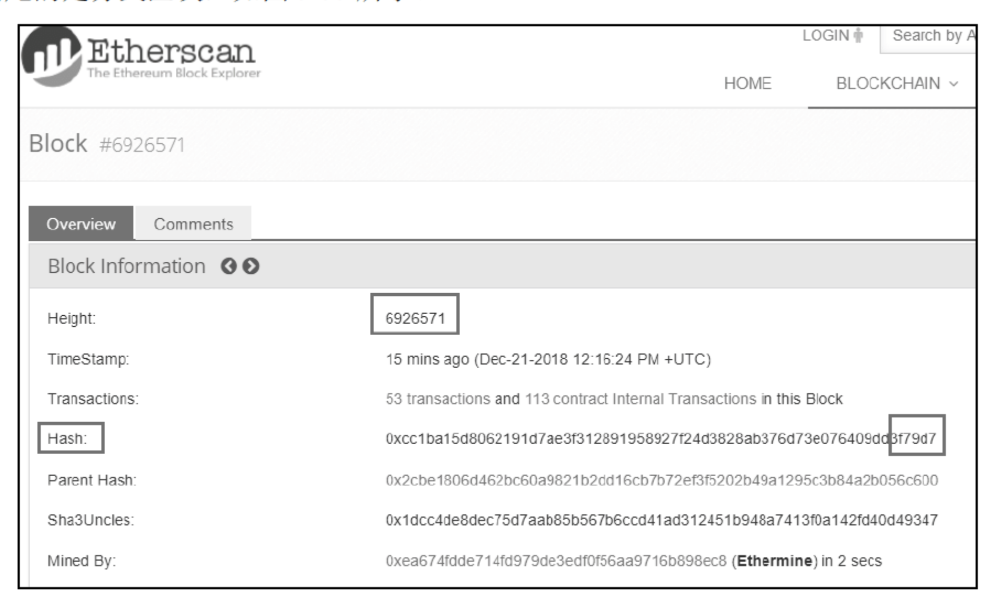



图 5-55　到区块链浏览器查询以验证区块扫描出的分叉区块是否正确

同时查看“eth_block”数据表，验证是否已经将分叉的区块标记为分叉状态，图 5-56 所示是已经成功将哈希值结尾为“54”的“6926571”号区块标记为分叉区块的情况。

此外，我们再通过 MySQL 命令来查看“eth_transaction”数据表中是否存储了交易信息（见图 5-57）。具体命令是：

select * from eth_transaction limit 5;　// limit 5 的意思是只选出 5 条，因为交易信息比较多。

select * from eth_block;

| block_number | block_hash | parent_hash | create_time | fork |
|---|---|---|---|---|
| 6926560 | 0xd23b42d6bcbb462f403e631684478ce8bcdd437fbb790c58ff66a315ebb13fad | 0x49d4cd0e4c8b6e7a173c7fab81d0e701563b1c2a606ef984ced00cb9d72037c5 | 1545394475 | 0 |
| 6926561 | 0x1c7876e7b4d1be07919db4a3c882441d7413b93cec35a7d10f6454c46d798a10 | 0xd23b42d6bcbb462f403e631684478ce8bcdd437fbb790c58ff66a315ebb13fad | 1545394482 | 0 |
| 6926562 | 0x019f9e23fbcbd07cfd255ab06e06b6d783a4bd19e745856e99df39a7d49f76e3 | 0x1c7876e7b4d1be07919db4a3c882441d7413b93cec35a7d10f6454c46d798a10 | 1545394498 | 0 |
| 6926563 | 0x88a3d7577ac72063e233b61d6b5c25df883c98b4133b0515d00c21d27dbe9f83 | 0x019f9e23fbcbd07cfd255ab06e06b6d783a4bd19e745856e99df39a7d49f76e3 | 1545394503 | 0 |
| 6926564 | 0xc392bd1f526a9ae1b4d7174c9f0014a45e6ac151cc981751a7c3f350b84a2741 | 0x88a3d7577ac72063e233b61d6b5c25df883c98b4133b0515d00c21d27dbe9f83 | 1545394506 | 0 |
| 6926565 | 0x997e704f5e4ada136ebbf88e46f8b506c209792cc3829a0b90d7ec92cce5ad1c | 0xc392bd1f526a9ae1b4d7174c9f0014a45e6ac151cc981751a7c3f350b84a2741 | 1545394516 | 0 |
| 6926566 | 0x9c323c9924466843fd060740bf4c2a25bd79bf47b1e8de754dd200e1385b1b5f | 0x997e704f5e4ada136ebbf88e46f8b506c209792cc3829a0b90d7ec92cce5ad1c | 1545394520 | 0 |
| 6926567 | 0x9f3b65d58c3ccbd74918a39b49a810f9c53f2918689b838bf36f155aecd97c01 | 0x9c323c9924466843fd060740bf4c2a25bd79bf47b1e8de754dd200e1385b1b5f | 1545394549 | 0 |
| 6926568 | 0xf4710ef1f6eba72c05f9a16b51cc5113bcc9d76709be17994017e120800a797c | 0x9f3b65d58c3ccbd74918a39b49a810f9c53f2918689b838bf36f155aecd97c01 | 1545394565 | 0 |
| 6926569 | 0xb7996973e39c89c3353d0b3229b6de1074a5e9c931594fe3d47b3caffce374c7 | 0xf4710ef1f6eba72c05f9a16b51cc5113bcc9d76709be17994017e120800a797c | 1545394567 | 0 |
| 6926570 | 0x2cbe1806d462bc60a9821b2dd16cb7b72ef3f5202b49a1295c3b84a2b056c600 | 0xb7996973e39c89c3353d0b3229b6de1074a5e9c931594fe3d47b3caffce374c7 | 1545394582 | 0 |
| 6926571 | 0x0532afedebf62e6e1500fd5daefc9163c4c5d9291899fb477c3e590f4aa5ee54 | 0x2cbe1806d462bc60a9821b2dd16cb7b72ef3f5202b49a1295c3b84a2b056c600 | 1545394584 | 1 |
| 6926572 | 0x46e45e38db0620cd984114a01045176bfa953f0f7ce39e8b728b0e6131aeef56 | 0xcc1ba15d8062191d7ae3f312891958927f24d3828ab376d73e076409dd3f79d7 | 1545394620 | 0 |
| 6926571 | 0xcc1ba15d8062191d7ae3f312891958927f24d3828ab376d73e076409dd3f79d7 | 0x2cbe1806d462bc60a9821b2dd16cb7b72ef3f5202b49a1295c3b84a2b056c600 | 1545394584 | 0 |
| 6926573 | 0x4aa2a705ef343bd93aa8c30009a71c68dbf0d660f99179c1afdaab0a94fb8584 | 0x46e45e38db0620cd984114a01045176bfa953f0f7ce39e8b728b0e6131aeef56 | 1545394629 | 0 |
| 6926574 | 0x01062731c1ea4447e9f8366d79fc23797538aace2c9af17612f6257b4a37bba6 | 0x4aa2a705ef343bd93aa8c30009a71c68dbf0d660f99179c1afdaab0a94fb8584 | 1545394651 | 0 |

图 5-56　在数据库控制台查看分叉区块是否被正确地打上了标记

```
mysql> select * from eth_transaction limit 5;
| id      | hash                                                                | nonce      | block_hash
          | to                                         | value              | gas_price    | gas      | input
| 20917   | 0x6d63f9dc30a3ce927f9ac8c5c58bc0a78b9006592dab1dd876b4c6a288c470cc | 0x1242     | 0x568fd71cddc2843c3e22
f5fa619   | 0x8cd647dae5138848ebda91b9c05de30722c2865e | 0xa53e6ae5cb000    | 0x4800000000 | 0x5208   | 0x
| 20918   | 0x07df98e87d7b4a22305db1e6c999afbf1af664c0629195594cb8241945b86d22 | 0x1d59     | 0x568fd71cddc2843c3e22
7e0b5f2   | 0x027282a8d16c26b6dc28552d3d3727a2eb332515 | 0xa53e6ae5cb000    | 0x4800000000 | 0x5208   | 0x
| 20919   | 0xe974bb43b19b68260e5d3dc31224993251a904206ff7a6b5b863092096a663bc | 0xb8       | 0x568fd71cddc2843c3e22
d9b4cf4   | 0xf629cbd94d3791c9250152bd8dfbdf380e2a3b9c | 0x0                | 0x174876e800 | 0xcb9c   | 0xa9059cbb
a7df97c0000 |
| 20920   | 0x2e52c38f2951f56d0670817df7db9a9c3e231a4d1cdc7f94f37ed62f606336c2 | 0x1        | 0x568fd71cddc2843c3e22
38cac44   | 0x7b5bcf82fabbbb1853b8d2d05086b6379e1ac02b | 0x6737139e3e800    | 0x12a05f2000 | 0x5208   | 0x
| 20921   | 0x4954625b3dfc42ffb6ab4605fbd1bbc6a2b7b6ec0b18709fb5148cfd2b394b5e | 0x16484    | 0x568fd71cddc2843c3e22
a87ab5f   | 0x1cb085521c11dfdef3103224952d268cbcc17f0f | 0x6ef3f224033f8000 | 0x1087ee0600 | 0x15f90  | 0x
```

图 5-57　在数据库控制台查看交易保存表是否有数据

由图 5-57 可以看到，交易信息完整地被存储了下来。其中的“input”字段就含有交易的参数数据，进一步分析它，可以实现很多需求的功能。

至此，整个区块遍历器测试通过。

## 5.3.5　理解监听区块事件

在实现了区块扫描器后，我们已经能够从区块中成功地获取到每笔交易的数据。

我们前面提到的要实现的监听区块事件，包含“transfer”等，主要就是从交易数据的“input”字段中解析出来的，“input”其实就是“data”。我们知道，“data”字段的前 10 个字符是由智能合约的函数名称转化而来的“methodId”，这就意味着，从“input”提取前 10 个字符来和对应函数的“methodId”进行对比，就能找出当前交易所调用的智能合约函数，从而实现事件的监听。与此同时，对应函数的参数就是“input”后面的部分数据，对应转换即可。

例如，我们知道“transfer”的“methodId”是“0xa9059cbb”，此时遍历区块的交易数据时获取到一笔交易的“input”数据：

```
0xa9059cbb0000000000000000000000003faa6c0794b47100aaef42ea93cc03e3f1c991f7
000000000000000000000000000000000000035af1c0cba270aea800000
```

截取上面这个“input”的前 10 个字符，即“0xa9059cbb”，再与“transfer”的“methodId”对比，发现是一样的，即证明这笔交易调用了智能合约的“transfer”函数，是一笔转账交易。

图 5-58 所示是从交易数据中解析出监听事件的整体流程图。

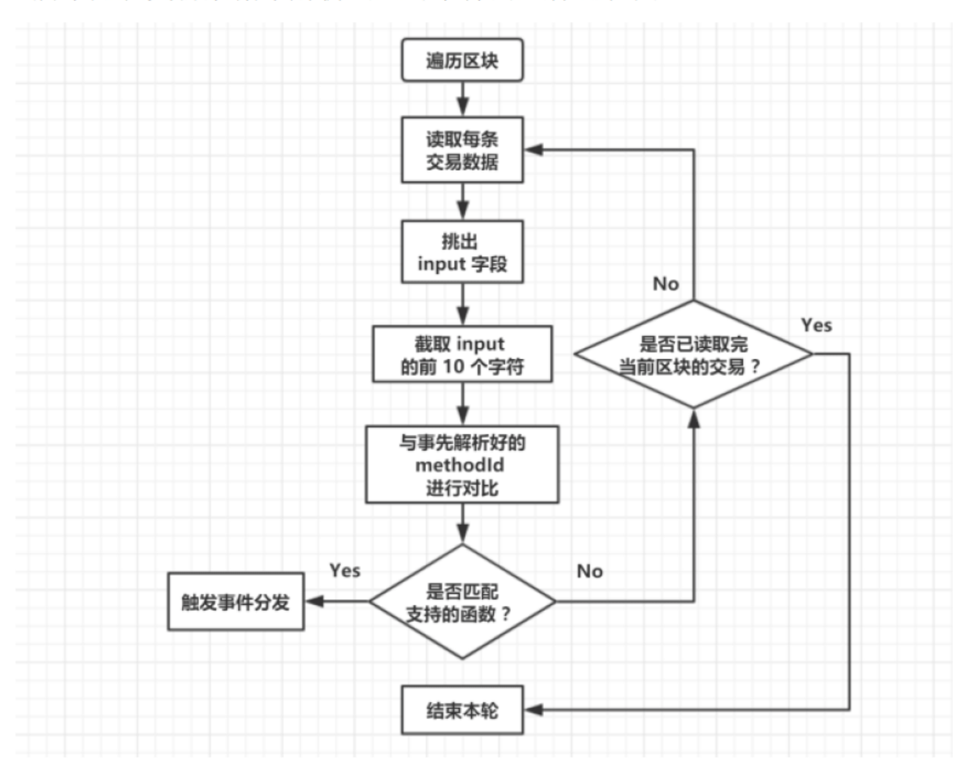



图 5-58　从交易数据中解析出事件的设计流程图

# 5.4　小　结

本章使用第 4 章中所实现的绝大部分以太坊 RPC 接口请求函数，采用 Go 语言实现了一个强大的中继服务支持程序。

本章内容也可以说是前几章内容的综合运用，其中还介绍了如何通过编程实现区块的分叉检测，同时拓展性地讲解了分叉区块数据的存储回滚。

由于以太坊中继是偏向于后端服务的，因此在整体实现过程中涉及了 MySQL 数据库的知识以及一些后端框架的使用，但是并不复杂。在真实的业务场景中，中继内部还可以加入其他中间件服务，比如将区块解析的事件存储到消息队列中，让其他微服务读取或使用等，以实现更复杂的功能。读者可自行尝试。

# 后　记

由于工作繁忙，因此编写此书的大部分时间是在夜间和周末。从开始到结束，深感写书之不易。这是我的第一本书，对于书中的每一字句都负有不可推卸的责任，如若书中还有内容未能做到深入细致，还请读者多多包涵，并且欢迎大家和我交流。

最后，衷心祝愿每一位阅读此书的朋友都能有所收获。

**引用**

此书的完成离不开以下文章的帮助（排名不分先后）：

https://blog.csdn.net/chabuduoxiansheng1/article/details/79740018

https://blog.csdn.net/s_lisheng/article/details/78022645

http://blog.luoyuanhang.com/2018/05/02/eth-basis-block-concepts/

https://blog.csdn.net/wo541075754/article/details/79042558

https://blog.csdn.net/ggq89/article/details/80072876

https://blog.csdn.net/weixin_37504041/article/details/80474636

https://ethereum.gitbooks.io/frontier-guide/

http://www.nahan.org/2018/10/02/utxovseth%EF%BC%9F/

https://ethfans.org/toya/articles/588

**本书源代码下载**

请访问以下地址下载本书源代码：

https://github.com/af913337456/eth-relay

**致谢**

感谢在我完成这本书的过程中给予建议或解答问题的朋友：远航，王金柱，陈东伟，陈婷，彭智锦（排名不分先后）。

**联系方式**

QQ 群：484707527

个人 QQ：913337456

邮箱：913337456@qq.com

技术博客：http://www.cnblogs.com/linguanh/

GitHub：https://github.com/af913337456/

林冠宏

2019 年 3 月